1972

An abstract machine to control the execution of semi-independent concurrent computations

Virgil Eugene Wallentine
Iowa State University

INFORMATION TO USERS

This dissertation was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

University Microfilms

300 North Zeeb Road
Ann Arbor, Michigan 48106

A Xerox Education Company

An abstract machine to control the execution of semi-independent concurrent computations

by

Virgil Eugene Wallentine

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Major: Computer Science

Approved:

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1972

TABLE OF CONTENTS

CHAPTER I.

INTRODUCTION AND OVERVIEW

In recent years the emphasis placed on providing a computer utility which accommodates many concurrent users has given rise to many new problem areas. The basic premise of such a computer utility -- that the system provide users with:

1) an efficient environment for program development, debugging, and execution;
2) a wide range of problem solving facilities;
3) low-cost computing through the sharing of resources and information --

has given impetus to the development of computer systems which have the following common characteristics (4):

1) concurrency of (parallel) activities,
2) automatic resource allocation,
3) sharing - the simultaneous use of a resource by more than one process (computation),
4) multiplexing of resources (mutually exclusive access by a computation to a resource for an interval of time),
5) remote conversational access by users,
6) nondeterminacy (unpredictable ordering of events),
7) long-term storage.

There are many interesting questions to be answered in each of the above areas of which the following is but a sample.

1) Does such a system perform the functions expected of it?
2) Can the system be extended or contracted by the addition or removal of resources?
3) Is it possible for the system to "die"?
4) Can a system's demise be circumvented?
5) What control mechanisms are necessary to permit user intercommunication?

Unfortunately, in many cases, the state of the art is not such that we can find definitive answers to these questions. A theoretical foundation has not yet been laid within which these questions can be posed in order to receive precise answers.

While the study of programming languages has benefitted greatly from various formal models of programming (see for example, Landin (15), McCarthy (19), and Lucas, Lauer, and Stigleitner (18)), no corresponding formal model has emerged to aid in the study of computer systems. Some models, such as Petri nets (21) and flow graph schemata (22) have been investigated with encouraging results, but such models have not yet been shown to realistically represent modern computer systems let alone provide insight for future systems.

This dissertation investigates the Vienna Definition Language (18) as a candidate for modeling computer systems. The Vienna Definition Language was originally developed for the formal definition of PL/I; however, it has proven to be generally applicable for language definition and has been used to formally define several programming languages. It is this author's belief that the Vienna definition method can also be applied

to the formal definition of computer systems.

The Vienna method is described by Wegner (23) and Lucas, Lauer, and Stigleitner (18). It consists of a language in which to describe operations on a data structure and an abstract machine which interprets all statements in the language. This machine is characterized by the set of states it may attain and a state transition function. The initial state of the machine is defined in terms of a given program, and the subsequent behavior of the machine is said to define the interpretation of the program. The abstract machine therefore attaches a meaning to programs and their constituent parts by defining the effect of their interpretation on the state of the machine.

In this dissertation we investigate the use of the Vienna method in formalizing one aspect of computer systems, namely the control of concurrent computations. The basic approach taken is to design an abstract machine whose structure is simple, yet sufficient to define the execution of parallel activities. The representation of the state of the machine coupled with the state transformation function provide a simple framework within which to study computer systems. Although more complex and more efficient machines can be designed, the necessary mechanisms for a concurrent control machine are presented in this simple model.

The flow of control during execution of a program emanates from the control structures available to the programmer. Thus, in Chapter II we investigate the control structures of current programming languages with particular emphasis on control structures useful for controlling concurrent computations. The semantics of the control structures are described

informally and examples of their use are presented.

In Chapter III we develop the architecture of the abstract machine. The information structure representing the state of the machine is defined in terms of the information accessible to all computations (common data through which the computations communicate) and that information which is local to each computation (data which affords a computation its independence). Transformation of the state of the machine is effected by the execution of an instruction in one of the currently active computations. The skeleton of a concurrent computations machine is completed with the specification of the machine's initial state, the structure of all final states, and the nondeterministic manner in which the computations and instructions within computations are chosen for execution.

In Chapter IV we abstract from the informal descriptions of Chapter II that set of primitive control mechanisms sufficient to implement the control structures presented in Chapter II. Control primitives are defined for the initiation of computations, the termination of computations, and the synchronization of computations.

In Chapter V the applicability of the machine and the set of task control primitives presented in Chapters III and IV is illustrated. Two simple programming languages which accommodate concurrent computations are given and are defined using the methods of Chapters III and IV.

In Chapter VI we discuss the strengths and weaknesses of the abstract machine as a framework in which to imbed the study of computer operating systems. Particular emphasis is placed on its conceptual simplicity.

CHAPTER II.

CONTROL FEATURES OF MULTITASKING PROGRAMMING LANGUAGES

The motivation for the specification of parallel computations in a program is not so much to make a particular program execute more efficiently as it is to relax the constraints on the order in which parts of the program are executed. A scheduling algorithm is therefore allotted more degrees of freedom in the ordering of executions and more freedom to effect greater efficiency in the allocation of resources. In this chapter we informally examine the control structures proposed in the literature which permit programmers to specify concurrent operations.

In order to make clear the terminology used, the following definitions are given. Control structures are the operations which specify the sequencing rules for programs or parts of programs. Computation will be used to indicate the sequence of instructions, specified by the programmer, the interpretation of which will produce the intended results. Concurrent computations are those which can be interpreted simultaneously and produce no unintended results. In some cases the order of interpretation of computations is dictated by inter-computation dependencies. The term semi-independent means that computations can be interpreted as independent entities except at certain points (where they are dependent on one another) of explicit communication.

The term task[1] refers to the data structure - a set of cells containing data and its structural relationships - containing all the information

---

[1] A synonym for task will be process.

necessary to perform the computation. It consists of the information accessible to a computation and the computation itself. A computation can be interrupted, blocked, and resumed at a later time as long as all the task information is saved. The control of concurrency of computations is affected by the synchronization of transformations.

Control structures of current multitasking programming languages are surveyed in order to identify the common properties among them. In order to gain some insight into the mechanisms necessary to implement control structures, we present in this chapter an informal description of the actions effected by the following control structures:

1) transfer statements,
2) procedure call,
3) block entry,
4) block/procedure exit,
5) task initiation,
6) task termination,
7) task synchronization.

The first four items on the above list are common to single task languages and are well understood. Since a task consists of a set of accessible cells and an instruction sequence representing a computation, the execution of a transfer statement effects a change of the instruction sequence to be executed and a change in the names and number of memory cells accessible to the task. Both items are associated with the label specified as the operand of the transfer statement.

Execution of a procedure call effects a change in both components of a task. The new instruction sequence is specified in the declaration of the procedure whose identifier is the operand of the call. The set of cells accessible to a procedure during execution is the union of the set specified at procedure declaration time and the set of cells associated with the actual parameters specified at the time of call. When a procedure body is activated by a call, some provision must be made to save the current components of the task. These components are to be restored upon procedure exit to enable the execution to be continued at the instruction following the procedure call.

The semantics of block entry only affect the cells accessible by the task. At entry, new cells (one for each identifier declared in the block) are created and the association between identifier and cell is established. If an identifier is declared which had a previous cell, the association is changed. Again the current task components must be saved for reinstatement at block exit.

Block and procedure exit instructions must, when executed, restore to the task those cells accessible prior to block entry and procedure call, respectively. No change is effected (other than the incrementation of the instruction pointer) to the instruction sequence on block exit. Procedure exit, however, must restore the instruction sequence which was effective at the time of procedure activation (i.e. procedure call).

The above four control structures have been formally defined by others (18) and there is general agreement on their usefulness. But there seems to be very little agreement on the control structures

necessary to specify the multitasking concept. Of the many control features cited in the literature, four types of synchronization control and three types of task creation and deletion are discussed. Most other control features of current programming languages are variations of the ones discussed.

One of the earliest proposed schemes for controlling concurrent computations was introduced by Conway (3) and later revitalized by Dennis and Van Horn (5). The basic control feature for parallel programming is specified by the statement

fork w; .

"w" is a label and "fork w" specifies a new task is to be initiated whose instructions are those at w and whose accessible memory cells are those accessible in the block in which w is declared. The termination of tasks is specified by

join t,w; ,

where "t" is a counter under programmer control specifying the number of tasks to be terminated before a task - the last one to complete its computation - executes a transfer to "w".

Communication between concurrent computations is affected by access to a common data base. The ability to change such a data base requires some mechanism to assure a particular task has exclusive access to the common data. An instruction

lock d;

effects such a lock on a data base where "d" is a variable lock indicator which is set to "on" if the data base is in use. The "lock d" sets "d" to "on" if not set. An instruction

unlock d;

sets the lock to "off".

A simple example of a program specifying concurrent computations is presented on the following page. This program specifies concurrent execution of the two computations

"a:=1", "c:=3"

and "b:=a+2"

followed by the single computation "d:=b+c". The intended value of "d" is 6 and the correct value is attained by synchronizing the computations by access to the common variable "a". It is set to "1" before the computation "b:=a+2" can proceed.

An informal description of its meaning follows. "t" is set to the value 2 in line 3 to indicate 2 computations are candidates to execute in parallel. A task is created in line 5 to set "a" to 1 and "c" to 3. The task executing line 5 continues to set "b" to "a+2" only after "a" has been set to "1" by the task assigned to execution of line 15. Note that the region of code which updates or checks the value of "a", which is the common data base, is protected by lock w. The concurrent

computations are

"b:=a+2"

and "c:=3".

Obviously "b" attains the correct value of 3 if and only if "a" is set to 1 first. One of the tasks will be assigned to the computation

"d:=b+c".

Only one task will be terminated since "t" has initial setting of 2.

```
 1.  begin  integer  t,a,b,c,d;
 2.         lock w; label wl, x, cycle;
 3.         t:= 2;
 4.         a:= 0;
 5.         fork wl;
 6.  cycle: lock w;
 7.         if a ≠ 1 then begin
 8.                       unlock w;
 9.                       goto cycle;
10.                       end;
11.         b:= a + 2;
12.         unlock w;
13.         join t, x;
14.     wl: lock w;
15.         a:= 1;
16.         unlock w;
17.         c:= 3;
18.         join t, x;
19.      x: d:= b + c;
20.  end
```

In another approach, Dijkstra (6) has proposed a set of control structures for specifying concurrent computations. His extensions to Algol include a sequence of n concurrent computations surrounded by the special statement bracket pair "parbegin" and "parend". This is interpreted as parallel execution of the constituent statements. He also presented two synchronization features which operate on special variables called "semaphores" which are initialized but not referenced by any operations other than the special task control operations "P(semaphore)" and "V(semaphore)". A program using these control structures and specifying the same concurrent computations as in the previous example is presented.

```
1.  begin  integer  a,b,c,d
2.         semaphore sem; label cycle;
3.         sem:= 1;
4.         a:= 0;
5.         parbegin
6.            begin
7.            cycle: P(sem);
8.                   if a ≠ 1 then begin
9.                                 V(sem);
10.                                goto cycle;
11.                                end;
12.                  V(sem); b:= a + 2;
13.           end
14.           begin P(sem); a:= 1;
15.                 V(sem); c:= 3;
16.           end
17.        parend
18.        d:= b + c;
19. end
```

The V-operation increases the value of its argument semaphore by 1 and it is considered an indivisible operation. The P-operation functions to decrease the value of its semaphore argument by 1 as soon as its resulting value will be non-negative. The completion of the P-operation - i.e. the decision that this is the appropriate moment to effectuate the decrease and the subsequent decrease itself - is also considered an indivisible operation.

Using the same task initiation and termination control features as Dijkstra, Hansen (9) has recently introduced more natural (in the author's opinion) synchronization features. Shared variables are referenced only within critical regions. The statement

region v do S

indicates that statement "S" is executed whenever the current task owns the shared variable "v". The conditional critical region S, indicated by

region v when B do S

is executed when the additional restriction, that condition B is interpreted as true, is met. Again, the same program is specified.

```
 1.  begin  var a,b,c,d: integer;
 2.         var v: shared lock;
 3.         a:= 0;
 4.         parbegin
 5.            region v when a do b:= a + 2;
 6.            begin region v do
 7.                        a:= 1;
 8.                        c:= 3;
 9.            end;
10.         parend;
11.         d:= b + c;
12.  end
```

The tasking facilities of PL/I (12) and Fitzwater and Schweppe (7) specify the activation of procedures as tasks. The initiation of a task is a procedure call,

call proc (arg(1),---,arg(n)) task t, event e;

which specifies also an event variable. The state of the event variable is implicitly set upon completion of the assigned task. Task completion is implicit upon completion of the procedure. The synchronizing control is provided in the statements signal (event) and wait (event). These statements set the event variable to "happened" and release the task, from a blocked list for the specified event, respectively. Again, the same concurrent computations are specified.

```
 1. begin  integer a,b,c,d;
 2.        task t; event e1, e2;
 3.   procedure pr(i); integer i;
 4.        begin  a:= i;
 5.               signal e1;
 6.               c:= 3;
 7.        end
 8.   begin
 9.        a:= 0
10.        call pr(1) task t, event e2;
11.        wait(e1);
12.        b:= a + 2
13.        wait(e2);
14.        d:= b + c;
15.   end
16. end
```

Many other control features have been proposed, but their inclusion would be of little additional value since most of them are combinations of the above. The interested reader may consult the task control structures of Oregano (2), SOL (13) (14), Algol 68 (17) or the control structures presented in a paper by Anderson (1) for further descriptions.

CHAPTER III.

THE CONTROL OF CONCURRENT COMPUTATIONS

This chapter presents the architecture of a class of abstract machines, called CONCOMs, whose structure is sufficient to control the execution of concurrent computations. Description of both the machine structure and the manner in which the structure is transformed by execution of instructions is presented using the Vienna Definition Language (18). An outline of the chapter is as follows:

1) The structure of the state of a CONCOM is presented.
2) The transformation of a state to the next state of a CONCOM is given.
3) And finally, the initial and all final states of a CONCOM are specified.

Complete definition of a CONCOM is accomplished by specifying the interpreter instructions whose execution effects the state transformation. This chapter thus presents the framework within which to define concurrent computation interpreters. The complete definition of such interpreters will be deferred until Chapter V.

The definition of a CONCOM machine will be given in terms of the objects from which the state of the machine is built, the structure of the objects, the manner of referencing the objects, an initial state, any final states and a state transformation function. In particular, we have the following definition of a CONCOM.

Definition 3.1

A concurrent computations (CONCOM) interpreter is a 6-tuple of sets and operations

(EO, SEL, is-state, IS, FS, TF)

in which:

1) EO is a nonempty set of elementary objects;
2) SEL is a nonempty set of selectors;
3) is-state is a predicate defined over a set of objects O built from the sets EO and SEL;
4) IS is a special object, satisfying the predicate is-state, called the initial state;
5) FS is a set of objects, each satisfying is-state, called final states;
6) TF is a state transition (unary) operator whose argument satisfies is-state and which takes its values from the set of objects which satisfy is-state.

O is the set of Vienna objects; IS and FS are elements of O; and TF is the transformation function whose argument and values are states. EO and SEL are Vienna elementary objects and selectors. Specification of the other four components of a CONCOM follows.

The data structure which defines the state of the machine is structured in two levels. The global (immediate components) level contains all the information known only to the interpreting machine. That is, it is the common data through which all tasks communicate. The second level -- task local level -- contains information known only to a particular task.

No other task can reference this data. It is this data which is saved when a task must suspend execution and which is restored when a task resumes execution.

The global level of the state of the control machine specifies

1) the attributes of values known to the tasks in the system,
2) the memory cells in use and their current values,
3) an element specifying the currently executing task,
4) a list of tasks currently ready for execution, and
5) a set of lists of tasks which are blocked and waiting for the use of a resource.

The immediate components of the state of a CONCOM are as follows:

1) a unique name generator which supplies the index of a new cell (unique name) accessible to a task when the interpreter instruction un-name is invoked. Its unique selector is "s-n";
2) an attribute table which defines the type of information contained in each memory cell. Its unique selector is "s-at";
3) a denotation table which defines the relationship between cells and their values. Its unique selector is "s-den";
4) a task selector which identifies the unique name of the task whose instruction sequence has been selected for execution. Its unique selector is "s-task";
5) a concurrent computations component which is a collection of all those data structures called tasks which are currently available for execution. Its unique selector is "s-cc".

Definition 3.2

The state of a CONCOM satisfies the predicate is-state defined as:

```
is-state = (<s-n:is-integer>,
            <s-at:is-at>,
            <s-den:is-den>,
            <s-task:is-name>,
            <s-cc:is-cc>).
```

Assuming a representative set of data types, the attribute table may contain designations for integer, boolean, label, and procedure variables. An additional data type is needed to specify the representation of a resource use lock variable. Each unique name of a cell selects from the attribute table the type of data contained in the cell.

Definition 3.3

The attribute table of a CONCOM has entries of the form <n:is-type> where

```
is-at = ({<n:is-type> | is-n(n)}),
is-type = INT V LOG V LABEL
          V PROC V LOCK,
```

n is the unique name of the cell, and is-n(n) tests the validity of n as a cell name of the form n_j (j is an integer).

The denotation table contains the values of the data types specified in the attribute table. The common data types such as integer and boolean have values as expected; but label, procedure, and lock variables have more interesting denotations. A label represents a new site of activity (new instruction sequence) and a new environment (new set of accessible cells). Furthermore, this new environment has an enclosing environment

(due to the accommodation of block structure) which must be restored upon block exit. The label denotation must therefore designate all three components.

Procedure denotations must designate the new instruction sequence, its environment, and its formal parameters. The components of a procedure denotation have values that are in effect at the time the procedure is declared. The formal parameters and instruction sequence are specified in the procedure declaration. The environment in which the procedure is executed is in effect at declaration time and is copied into the environment component of the procedure denotation.

Since more detail on the use of resource use locks will be presented in Chapter IV, very little discussion follows. The basic hypothesis, though, is that each resource will have an associated lock which is set to "on" when the resource is in use and "off" when not in use. Therefore, when a task cannot seize a resource it must be blocked and placed on a list associated with the resource. The lock will therefore have a lock and an associated list of tasks waiting for the resource. The lock denotation must designate both components. The structure of these tasks will be discussed later; and the denotation table definition follows.

Definition 3.4

The denotation table of a CONCOM has entries of the form

<n:denotation>

where

```
is-den       = ({<n:is-integer V is-log
                      V is-proc-den V is-label-den
                      V is-lock-den> | is-n(n)}),

is-proc-den  = (<s-attr:is-proc-attr>,
                <s-env:is-env>),

is-proc-attr = (<s-parm-list:is-id-list>,
                <s-st:is-st>),

is-label-den = (<s-st-list:is-st-list>,
                <s-env:is-env>,
                <s-d:is-dump>),

is-lock-den  = (<s-lock:is-log>,
                <s-ttab:is-ttab>),

is-ttab      = ({<n:is-task> | is-n(n)}),
```

and n is a unique cell name.

Note that in Definition 3.4 the "s-parm-list" selector references the parameter list of a procedure declaration and that the values of the actual parameters will be added to the execution environment at procedure call time. Also note that the selector "s-env" references that set of accessible cells via the declared identifier. "s-attr" selects the attributes of a procedure -- that is, the parameter list and statement list. The is-dump and is-task predicates are satisfied by task local components and are discussed later.

The "s-cc" selector retrieves those data structures called tasks. The tasks retrieved are those currently available for execution. That is, all tasks designated by a unique name in the concurrent computations component are on a ready list for execution. The next instruction to be executed can be selected from any one of the tasks on the ready list.

The ready list is really a task table since each task is selected by the unique name assigned to it at task initiation. Thus, assuming later definition of the predicate is-task, the concurrent computations component is defined.

Definition 3.5

The concurrent computations component of a CONCOM has entries of the form

<n:is-task>

where

is-cc = ({<n:is-task> | is-n(n) }).

The tree structure representing the immediate components of the state S is in Figure 3.1. And the tree structures representing the procedure, label, and lock denotations are shown in Figures 3.2, 3.3, and 3.4.

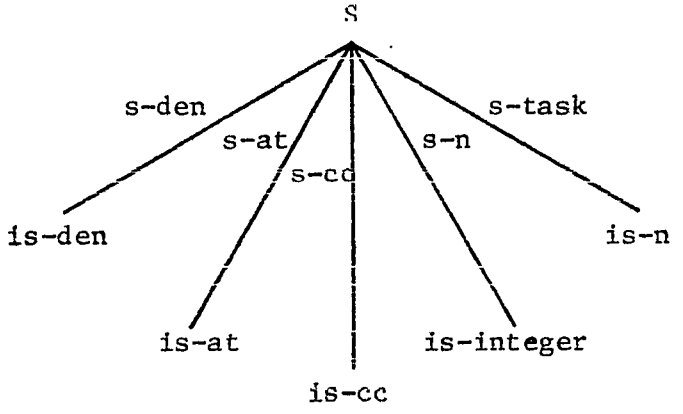

Figure 3.1. Immediate components of the state of a CONCOM

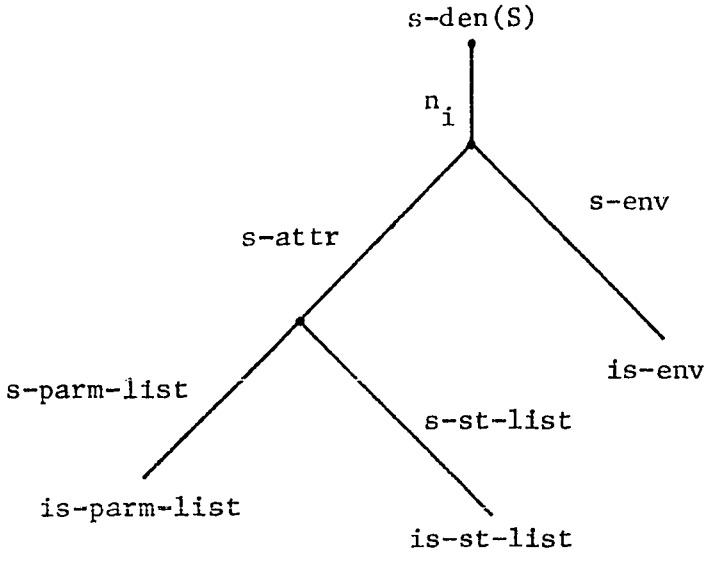

Figure 3.2. Procedure denotation for cell whose unique name is n_i

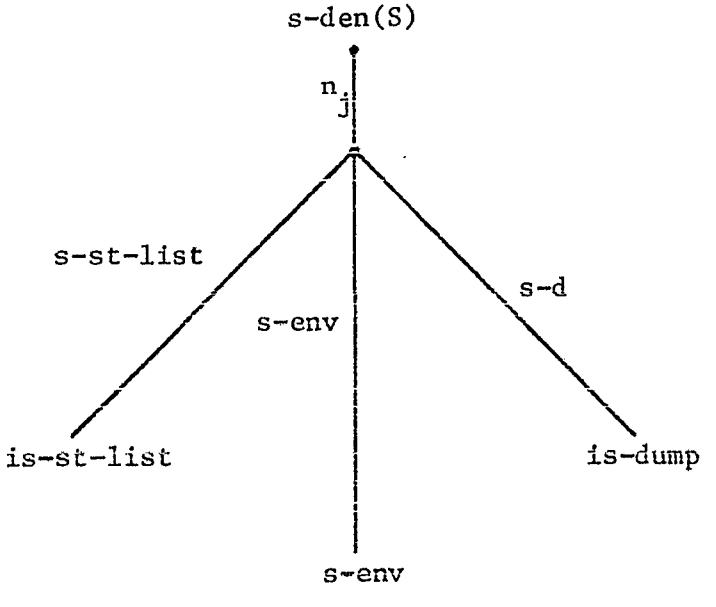

Figure 3.3. Label denotation for cell whose unique name is n_j

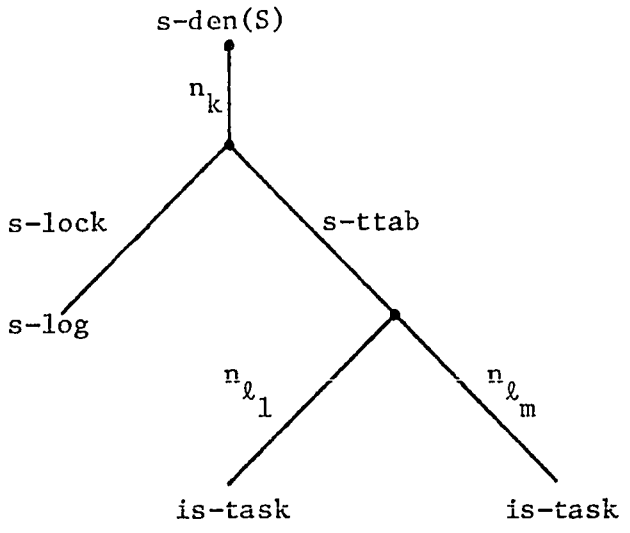

Figure 3.4. Lock denotation for cell whose unique name is n_k and which has m tasks in its task table

The task local information is that collection of data structures from which a task obtains its independent identity apart from all other tasks in the machine. Each task progresses in an independent manner except at explicit points of task intercommunication; and no assumptions of speed of other task execution are permitted. Therefore, a task will by necessity contain a sequence of instructions to be executed which represent the computation to be performed. It must also have access to that unique set of memory cells which contain values that are not accessible to other tasks and have access to memory cells common to other tasks. Furthermore, each task when executing has an enclosing environment of its

own which must be restored upon exit from the present environment. Hence the following definition of a task.

Definition 3.6

A task satisfies the predicate is-task, where

$$\text{is-task} = (\langle \text{s-c:is-c} \rangle, \langle \text{s-env:is-env} \rangle, \langle \text{s-d:is-dump} \rangle).$$

The environment of a task is specified by an environment table which defines the relationship between an identifier and the unique name of the cell containing its value. The selectors are identifiers and they retrieve the unique name of a cell. Every identifier referenced by the concurrent computation is a selector in the environment table of the executing task.

Definition 3.7

The environment component of a task has entries of the form

$$\langle x : n_i \rangle$$

where x is an identifier and n_i is a unique name. Therefore,

$$\text{is-env} = (\{\langle \text{id:is-n} \rangle \mid \text{is-id(id)}\}).$$

Many concurrent computations are specified in languages which have block structure and procedure declarations. And the execution of the instructions in a task may effect entry into and exit from blocks. The changing of the environment and sequence of instructions executed necessitate the saving of the environment component and the control component.

Therefore, a dump component is designated which contains both of these components plus a dump component which specified return information of the previously effective environment, control, and dump.

Definition 3.8

The dump component of a task satisfies the predicate is-dump.

$$\text{is-dump} = (\langle \text{s-c:is-c} \rangle, \langle \text{s-env:is-env} \rangle, \langle \text{s-d:is-dump} \rangle).$$

Each task is assigned a computation; and each computation is defined by a unique sequence of instructions. Therefore, the representation of a task must include a component containing a description of the program instructions to be executed. The control component is an object which is an element of the set O of composite objects which defines both the structure and sequence of instructions.

Instructions at a terminal vertex of the control tree (component) are available for execution; and the instruction at the terminal vertex chosen is deleted when executed. Execution proceeds to instructions at terminal vertices on the new control tree. When the only terminal vertex contains the null element Λ, then execution stops.

Even though use will be made of the macro format of instructions discussed by Wegner (23), the control tree contains the composite object representation of an instruction displayed in Figure 3.5. An instruction consists of three components, referred to as: the instruction name component selected by "s-in"; the argument list component "s-al"; and the

return information component "s-ri". Also associated with an instruction is a list of m possible successor nodes (further from the root of the tree) selected by "succ(1)",---,"succ(m)" which are instructions.

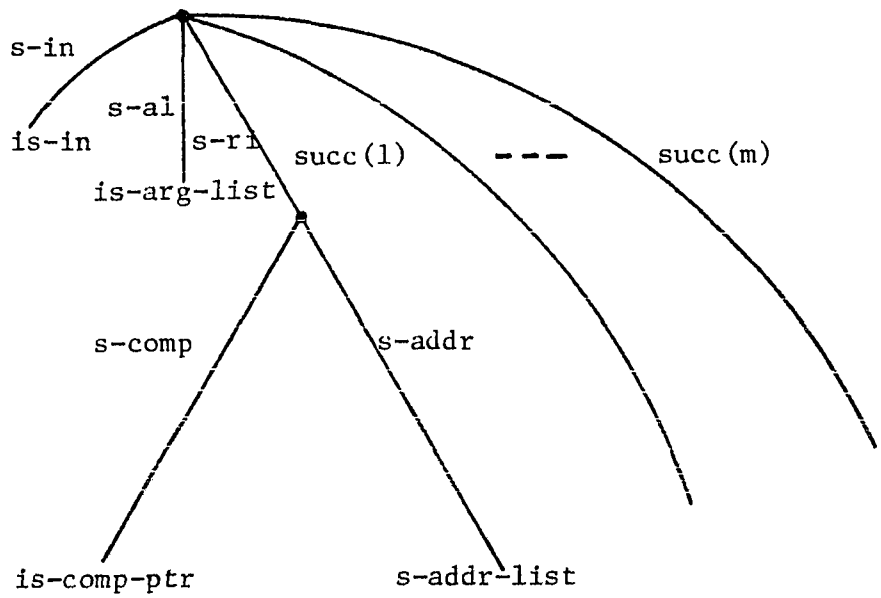

Figure 3.5. Structure of an instruction on the control tree

"s-al" selects the list of arguments of the instruction. These arguments are set to the values of value-returning instructions on successor nodes. Furthermore, a particular component of an argument may be selected. "s-ri" (return information) selects a two component structure which identifies the component selected for assignment by "s-comp"; its "s-addr" component indicates which arguments at which predecessor (closer to the root) nodes are to be assigned the value of this instruction. The following definition of the control component is recursive; but it is understood that the recursion is applied only a finite number of times.

Definition 3.9

The control component of a task satisfies the predicate is-c.

```
is-c = (<s-in:is-in>,
        <s-al:({elem(i):is-obj & ¬is-Λ> | is-integer(i)})>,
        <s-ri:(<s-comp:is-sel>,
               <s-addr:({<elem(i):is-arg-addr> | is-integer(i)})>,)>,
       {<r:is-c> | r ε SUCC})V is-Λ,
```

where

is-Λ is satisfied by the null object Λ;
is-in is satisfied by an interpreter instruction;
is-obj is satisfied by all elements of the set O;
is-sel ε SEL;
is-integer is satisfied by the set of positive integers;
SUCC = {succ(1),succ(2),--- };
is-arg-addr = (<s-arg:is-integer>,
<s-pred:is-integer>).

Note that the "s-arg" selects an integer i indicating the ith element of an argument list is to be set to the value of this instruction. The "s-pred" selects an integer j indicating the ith element of the argument list of the jth predecessor node is to be set to the value of this instruction.

Keeping the structure of the machine state in mind, a transformation of a state in our sequentialized machine is effected by the execution of an instruction in any task available for execution. Furthermore in order to allow the interpreter to describe all possible implementations some arbitrariness in the selection of instructions, as in the Vienna interpreter, in tasks is a necessity. Two levels of nondeterminism in selecting the interpreter instruction to be executed are next presented.

In a system which allows concurrent execution, we assume the processor is assigned to the task whose execution at this point effects the most efficient use of the machine. The task scheduler picks such a task from the ready list of tasks. In a CONCOM that set of tasks is associated with the concurrent computations component of the state. Therefore, each time an instruction is executed the scheduler may be invoked to select a task to be executed. Hence the definition of the set of task names of tasks available for execution is given here.

Definition 3.10

The set

$$TS(S) = \{n \mid is\text{-}n(n) \;\&\; \neg is\text{-}\Lambda(s\text{-}c \cdot n \cdot s\text{-}cc(S))\}$$

is called the set of task selectors of a given CONCOM.

The selection of an instruction for execution within a task is specified by a composite selector which selects the instruction at a terminal node of a task's control tree. This set of control selectors is defined below.

Definition 3.11

The set

$$CS(C) = \{ISEL \mid ISEL \in SUCC^* \;\&\; ISEL \cdot s\text{-}c \cdot n \cdot s\text{-}cc \neq \Lambda \;\&\; succ(i) \cdot ISEL \cdot n \cdot s\text{-}cc = \Lambda, 1 \leq i < \infty\}$$

is called the set of control selectors of a given control tree C in a task n, where SUCC* is the set of all sequences of composite selectors that can be constructed from the set SUCC of Definition 3.9.

In a CONCOM each of the instructions at a terminal vertex of the control tree of each of the tasks identified by the set TS of Definition 3.10 in a state S may give rise to a different next state. The set of next states associated with a given state S will be denoted TF(S). The transformation function TF (of Definition 3.1) is effectively the next state function of the nondeterministic abstract machine called a CONCOM.

The instruction execution cycle of a CONCOM may be defined by specifying the function TF, which may then be defined in terms of the set of state transition functions associated with the terminal vertices of the control trees of all available tasks. Letting the state transition function be indicated by ST(S,ISEL,n), where the current state is S, the instruction to be executed is selected by ISEL, and the task containing the instruction to be executed is selected by n, then the transformation function can be defined formally.

Definition 3.12

The transformation function is defined as an element of the set of state transition functions

$$TF(S) = \{ST(S,ISEL,n) \mid ISEL \in CS(s\text{-}c(n \cdot s\text{-}cc(S))) \ \& \ n \in TS(S)\}.$$

Since TF contains a state transition for every executable instruction in every available task, the execution of an instruction can be defined by specifying a selection rule for choosing an element of TF(S), together with a state transition function for individual instructions ST(S,ISEL,n). This selection rule is unspecified in the abstract machine since it corresponds to the algorithm for scheduling tasks. The state transition

function ST(S,ISEL,n) is defined in terms of the structure of the instruction -- illustrated in Figure 3.5 -- selected for execution.

Each terminal vertex of the control tree in task n has

1) an instruction component, in, selected by "s-in•ISEL•s-c•n•s-cc(S)";
2) an argument list, al, selected by "s-al•ISEL•s-c•n•s-cc(S)"; and
3) a return information component, ri, selected by "s-ri•ISEL•s-c•n•s-cc(S)".

The argument list for an instruction with m_{in} arguments has m_{in} components selected by elem(1)•al, elem(2)•al, ---, elem(m_{in})•al.

The state transition function (ST(S,ISEL,n) can be defined in terms of the following:

1) a function I_{in}, which depends on the m_{in} arguments of the instruction;
2) the state S with the current instruction vertex deleted;
3) the selector ISEL;
4) the selector n (task name); and
5) the return information component "ri = s-ri•ISEL•s-c•n•s-cc(S)".

Assume the instruction at the vertex ISEL•s-c•n•s-cc(S) has the following form:

$$\underline{\text{inst}}(x_1,x_2,---,x_m) =$$
$$p_1(x_1,x_2,---,x_m,S) \rightarrow a_1(x_1,x_2,---,x_m,S)$$
$$p_2(x_1,x_2,---,x_m,S) \rightarrow a_2(x_1,x_2,---,x_m,S)$$
$$\vdots$$
$$p_k(x_1,x_2,---,x_m,S) \rightarrow a_k(x_1,x_2,---,x_m,S)$$

where $p_1, p_2, ---, p_k$ are predicates and $a_1, a_2, ---, a_k$ are the actions associated with the predicates. The state transition can now be defined formally.

Definition 3.13

The state transition function is defined by

$$ST(S,ISEL,n) = I_{in}(elem(1)\cdot a1, elem(2)\cdot a1, ---, elem(m_{in})\cdot a1, \delta(S;isel), isel, ri),$$

where

$$I_{in}(x_1,x_2,---,x_m,S,isel,ri) =$$
$$p_1(x_1,x_2,---,x_m,S) \rightarrow a_1^*(x_1,x_2,---,x_m,S,isel,ri)$$
$$p_2(x_1,x_2,---,x_m,S) \rightarrow a_2^*(x_1,x_2,---,x_m,S,isel,ri)$$
$$\vdots$$
$$p_k(x_1,x_2,---,x_m,S) \rightarrow a_k^*(x_1,x_2,---,x_m,S,isel,ri)$$

and

$$isel = \text{"ISEL}\cdot s\text{-}c\cdot n\cdot s\text{-}cc(S)\text{"},$$

and

a_i^* is the state transition function associated with the action a_i specified by the system designer (interpreter programmer),

and

$\delta(S,isel)$ is the deletion operator of VDL which performs the function $\mu(S;\langle isel:\Lambda\rangle)$.

The state transitions a_i^* result from the execution of one of two kinds of instructions specified by a_i. If a_i is a macro instruction the new state, S_{new}, is obtained by replacing the "isel" component of the

state by a_i as follows:

$$S_{new} = \mu(S;<isel:a_i>).$$

In the case a_i is a value-returning instruction, its form is

$$\begin{aligned} \text{PASS:} \quad & e_0(x_1,x_2,---,x_m,S) \\ \text{s-sc}_1\text{:} \quad & e_1(x_1,x_2,---,x_m,S) \\ & \vdots \\ \text{s-sc}_\ell\text{:} \quad & e_\ell(x_1,x_2,---,x_m,S). \end{aligned}$$

The state transition a_i* involves evaluation of the expressions e_i, i=0,1,---,ℓ. The value "val(e_0)" is substituted in all addresses specified by ri and the values "val(e_i)", i=1,---,ℓ, are substituted for each component s-sc_i. If new_{tct} represents the control tree of the task being executed after the substitution of "val(e_0)" in all return addresses, after modification of the specified state components the new state is defined as follows:

$$\begin{aligned} S_{new} = \mu(S;&<\text{s-c.n.s-cc}(S):new_{tct}>, \\ &<\text{s-sc}_1:\text{val}(e_1)>, \\ &\vdots \\ &<\text{s-sc}_\ell:\text{val}(e_\ell)>). \end{aligned}$$

The new control tree of the task currently being executed is the only component of the new state which is yet to be specified. Introduction of a function "pred"[1], which operates on composite selectors

[1]This function is discussed by Wegner (23).

ISEL = $s_1 \cdot s_2 \cdot \text{----} \cdot s_k$ and removes the rightmost selector, is necessary. Thus, pred(ISEL) = $s_2 \cdot s_3 \cdot \text{----} \cdot s_k$. The function pred^i has been defined as the i-fold composition of pred. Thus $\text{pred}^i(\text{ISEL}) = s_{i+1} \cdot s_{i+2} \cdot \text{----} \cdot s_k$.

If the components s-arg and s-pred of s-addr(ri) are respectively i and j, the control tree position to which $\text{val}(e_0)$ is returned is defined by the following selector:

$$\text{s-comp(ri)} \cdot \text{elem(j)} \cdot \text{s-al pred}^i\text{(isel)} \cdot \text{s-c} \cdot \text{n} \cdot \text{s-cc(S)}$$

The selection process is as follows:

1) "s-c•n•s-cc(S)" selects the control tree of the currently executing task n;
2) "pred^i(isel)•s-c•n•s-cc(S)" selects predecessor node i of the current instruction being executed;
3) "elem(j)•s-al•pred^i(isel)•s-c•n•s-cc(S)" selects the jth element of the argument list of predecessor i;
4) "s-comp(ri)•elem(j)•s-al•pred^i(isel)•s-c•n•s-cc(S)" selects the component of the element of the argument list to which the value is to be assigned.

Definition 3.14

The next state S_{next} of a CONCOM attained after execution of an instruction in state S is defined by the VDL statement

$$S_{next} = \mu(S;\langle s\text{-}c\cdot n\cdot s\text{-}cc(S):new_{tct}\rangle,$$
$$\langle s\text{-}sc_1:val(e_1)\rangle,$$
$$\vdots$$
$$\langle s\text{-}sc_\ell:val(e_\ell)\rangle)$$

where two cases exist.

1) if a_i is a macro instruction then $\ell = 0$ and

   $new_{tct} = \mu(\delta(s\text{-}c\cdot n\cdot s\text{-}cc(S);isel);a_i)$

2) if a_i is a value-returning instruction then ℓ = number of expressions in the state transition function a_i* and

   $new_{tct} = \mu(\delta(s\text{-}c\cdot n\cdot s\text{-}cc(S);isel);$
   $\{\langle comp(i,j):val(e_0)\rangle \mid s\text{-}arg\cdot s\text{-}comp(ri) = j$
   $\& \; s\text{-}pred\cdot s\text{-}comp(ri) = i\},$

   where

   comp(i,j) is the selector
   $s\text{-}comp(ri)\cdot elem(j)\cdot s\text{-}al\cdot pred^{i}(isel)$
   $s\text{-}c\cdot n\cdot s\text{-}cc(S).$

The specification of a CONCOM is not complete at this point. The IS and FS components of the 6-tuple of Definition 3.1 have yet to be specified. IS is the initial state of a CONCOM. It specifies the contents of both global and local levels of the CONCOM before execution of any instructions. The set of final states, FS, contains the specification of the form of states $S_1, S_2, ---, S_m$ which have no succeeding state. That is,

$$TF(S_i) = \Lambda, \text{ for } i = 1,---,m.$$

The IS specifies one active task n_1 to execute a program. It initially has no accessible cells and no enclosing site of activity. Therefore, the denotation and attribute components of the state are null; and the environment and dump components of the task n_1 are null. But since a task n_1 is active, the current task component selects n_1; and the unique name generator component has value "2" so that no additional cell will have the unique name n_1.

Definition 3.15

The initial state of a CONCOM satisfies the predicate IS.

$$IS = \mu_0(\langle s\text{-}task{:}n_1\rangle, \langle s\text{-}n{:}2\rangle, \langle s\text{-}cc{:}\mu_0(\langle s\text{-}c{:}\underline{int\text{-}prog}(t)\rangle)\rangle),$$

where

int-prog(t) is an interpreter macro instruction whose expansion emanates the information structure transformations necessary to evaluate the program t to be executed.

The set of final states all have one common attribute. The control trees of all currently active tasks are null. Therefore, the scheduling algorithm will find no elements in the set of next states, because no state transitions will be specified.

Definition 3.16

The set of final states of a CONCOM is a collection of objects which satisfy the predicate is-final-state.

```
FS = {S | is-final-state(S)},
where
is-final-state(S) =
        (<s-den:is-den>,<s-at:is-at>,
         <s-n:is-integer>,<s-task:is-n>,
         <s-cc:{n:is-task | s-c•n•s-cc(S) = Λ}>).
```

CHAPTER IV.

CONCURRENT COMPUTATION PRIMITIVES

The control of concurrent computations emanates from the control structures available to the programmer. Of the control structures presented in Chapter II, techniques for implementing (and describing) transfer, procedure call, block entry, block exit and procedure exit are well understood and discussed elsewhere (18). It is the methods of implementing (and describing) the control of parallel activities (in the abstract machine) which are the concern of this and the succeeding chapter.

The implementation of control structures involves the establishment of an appropriate environment in which the instructions that perform the sequencing operations are to be executed. In the implementation of block structured languages, for example, a new environment must be established whenever a block is entered. In most block structured languages this establishment of a new environment can be described (and implemented) with the aid of a pushdown stack. The establishment of a new environment amounts to allocating (i.e. pushing) storage on top of the stack (together with the establishment of the proper linkage to items already in the stack).

In programming languages which accommodate parallel activities and in which environments may be shared among these parallel activities, great care must be taken when instructions interact with a common environment. In particular, in order that information not be incorrectly modified, certain precautions must be taken to insure that only one instruction modify a sharable environment at one time. Part of the guarantee that

this constraint is followed is obtained by assuming that certain instructions which control concurrent computations are uninterruptable. Such instructions will hereafter be referred to as primitive instructions. The main thrust of the current chapter will be a presentation of primitive instructions which initiate, terminate, and synchronize parallel activities (i.e. tasks).

Initiation of a parallel activity involves the creation of a special data structure called a task, followed by its placement on a special list called the ready list. In a CONCOM, the creation of a task is accomplished by creating a tree consisting of an environment component, a dump component, and a control component. This tree is assigned a unique name and attached as a subtree on the concurrent computations component of the state of the CONCOM. Placement on this component, selected by "s-cc", amounts to adding this task on the ready list. (Note that the initiation of a parallel activity results in the creation of a unique task with its own unique name. If the newly created task is to be associated with an identifier, this association must, of course, be made in some appropriate environment.)

Since the interruption of a task's creation and placement on the ready list might result in incorrect data structures, these actions must be made uninterruptable. Thus, a special primitive instruction "init-task" is defined to perform these functions. The execution of the instruction init-task(arg1,arg2,arg3,arg4) generates a new branch on the "s-cc" component of the machine state S. The arguments for the init-task

are defined as follows:

arg1 - the unique name of a task which is created for each task initiation;

arg2 - the data structure consisting of the cells accessible to the newly created task which are created or copied from an existing environment;

arg3 - the control tree (which is created in the form specified in Definition 3.9) containing the interpreter instructions which are to implement the specified computation;

arg4 - the dump component which is created (in the form specified by Definition 3.8) or copied from a denotation.

Definition 4.1

The primitive function which initiates a new task in a CONCOM is defined by the VDL instruction init-task.

init-task(n,CON,ENV,DUMP) =

s-cc: μ(s-cc(S);<n:μ_0(<s-env:ENV>,
<s-c:CON>,
<s-d:DUMP>)>).

where

S is current machine state,

n is the unique name of the task being created,

CON is the control tree of the task being created,

ENV is the environment in which the task is to be executed,

and DUMP is the stack of task local composite objects
μ_0(<s-c:is-c>,<s-d:is-d>,<s-env:is-env>)
which defines the enclosing site of activity.

The action of terminating a task in a CONCOM is accomplished by deleting from the "s-cc" component of the state that tree identified by the task's unique name. In VDL this is accomplished by creating a new "s-cc" component which contains all tasks but the task to be terminated. Then when a task is selected for execution, from the set TF(S), the control tree of this task is null and not available for execution.

A word of caution is in order regarding the termination of tasks. If the task to be terminated is the only active task then the interpretation of the entire program will be stopped prematurely by termination of the task. Normally, the intent in such a case is that this (solely active) task be assigned to continue the ensuing (parent) computation. To accommodate this situation a counter is maintained which keeps track of the number of currently active tasks and is used in task termination.

Termination is accomplished by the execution of the CONCOM primitive terminate (arg), where arg is the count of the number of active tasks. It performs a decrement of the count; and if it then has value 1, this task is assigned the ensuing computation by default. If not 1, then this task is deleted from the ready list. Note the necessity for uninterruptible execution between decrementing the counter and removable from the list. terminate can now be defined formally.

Definition 4.2

The primitive function which terminates m-1 tasks in a CONCOM is defined by the VDL instruction terminate.

terminate(ctr) =

(ctr•s-den(S)≤1) →

s-den: μ(s-den(S);<ctr:ctr•s-den(S)-1>)

T →

s-cc: μ(Λ;{<n:n•s-cc(S)> | n≠s-task(S)})

s-den: μ(s-den(S);<ctr:ctr•s-den(S)-1>),

where

"ctr" is the denotation of the count of the number of currently assigned tasks (initial value is m),

and "s-task" is the CONCOM state component which designates the task executing this primitive.

Communication between tasks in a CONCOM is provided by access to common variables. Tasks n_1 and n_2 can access common variables $v_1, v_2, ---, v_m$ if the "s-env" component of both n_1 and n_2 has selectors $v_1, v_2, ---, v_m$ and the unique names (cells) assigned to $v_1, v_2, ---, v_m$ are the same in both n_1 and n_2.

Since tasks can have access to common cells, each time a task wishes to send a message it merely sets the value of a common cell to a predetermined value. The receiving task needs only to check the value of the cell. If it has attained the agreed upon value, the message has been set and execution proceeds. If the cell does not contain the predetermined value, the receiving task must loop on the check until the value is attained.

But looping on the test, called a "busy" test, is inefficient; and no guarantee can be given that access to the common cells is mutually exclusive. That is, the value may be changed by some other task between the time it is tested and the time it may be set again, possibly to another predetermined value to indicate the receipt of the message.

Therefore, in order to provide mutually exclusive access to a cell, it is treated just like any other resource. Each resource has an associated use lock. All tasks which have access to the common cell must agree to synchronize their use of the common cell according to the following rules.

1) Each task must set the lock before referencing the cell. If the lock is already set, the task must block itself until the lock is reset.
2) Each task must not leave the lock set indefinitely. The task will reset the lock as soon as its exclusive access restriction is relaxed.

This mechanism provides for exclusive access to cells between the setting and resetting of the lock.

Note that the setting of a lock is really a request for a resource, and it must perform two functions. It must set the lock if it isn't presently set. If it is set, the current task must be blocked and remembered so that it may be resumed when the lock is cleared. Using the structure of a lock from Definition 3.4, the block of a task is indicated by placing it in the table of tasks ("s-ttab" component of a lock denotation) associated with this lock. It must then be deleted from the ready list, since it cannot be executed. Since interruption between the steps of testing of a lock, setting it, and moving a task to a blocked list may cause incorrect synchronization, the requesting of a resource is a primitive action and is accomplished by the primitive, request (lock), specified in Definition 4.3.

Definition 4.3

The primitive function necessary for a task to request common resources is defined by the VDL instruction request (lock).

request (lock) =

```
(s-lock•lock•s-den(S)) →
  s-den:  μ(s-den(S);
           <s-ttab•lock:s-task(S)•s-cc(S)>)
  s-cc:   μ(Λ;
           {<n:n•s-cc(S)>
              | n≠s-task(S) & is-task(n•s-cc(S))})
                    T →
  s-den:  μ(s-den(S);<s-lock•lock:"on">),
```

where

"lock" is the unique name associated with the pertinent resource use lock.

If it is again observed that the setting of a lock is a request for a resource, then upon the release of a resource (resetting a lock) all tasks currently waiting for the use of the resource become candidates to seize the resource. It is natural then that a scheduler be associated with each category of resources (and thus their associated locks). Each scheduler then may pick from the set of blocked tasks, BT (lock),

$$BT(lock) = \{n \mid n \bullet s\text{-}ttab \bullet lock \bullet s\text{-}den(S) \neq \Lambda \;\&\; is\text{-}task(n \bullet s\text{-}ttab \bullet lock \bullet s\text{-}den(S))\}.$$

Each task whose name n is an element of the set, $n \in BT(lock)$, is a candidate to obtain the resource.

Just as the requesting of a resource is a primitive action so is the releasing of a resource. The release (lock) instruction of Definition 4.4 specifies the action of the interpreter in resetting a lock or scheduling for execution one of the tasks on the blocked list for the resource use lock "lock". The function sel(set) of Definition 4.4 is left unspecified in this definition as it is the selection function on the set "set" which corresponds to the scheduling algorithm for the pertinent lock. Again, note that incorrect execution may occur if release is interrupted between the testing of the lock and movement of the task from the blocked to the ready list.

Definition 4.4

The primitive function necessary to allocate resources to competing tasks is described by the VDL instruction release (lock).

```
release (lock) =
   is-Λ(s-ttab•lock•s-den(S))
     → s-den:  μ(s-den(S);<s-lock•lock:"off">)
   T →
       s-cc:  μ(s-cc(S);
                {<n:sel(BT(lock))•s-ttab•lock•s-den(S)>
                   | is-task(n•s-ttab•lock•s-den(S))
                   & n = sel(BT(lock))}
       s-den:  μ(s-den(S);
                 {<n•lock:Λ>
                    | is-task(n•s-ttab•lock•s-den(S))
                    & n = sel(BT(lock))}
```

where

"sel" is a VDL primitive function which selects
a unique task name, from the set BT (lock),
whose task is to be scheduled for execution;
BT(lock) = {n | n•s-ttab•lock•s-den(S)≠Λ
& is-task(n•s-ttab•lock•s-den(S))};

and "lock" is the denotation of the resource use lock
of the requested resource.

CHAPTER V.

APPLICATION OF THE ABSTRACT MACHINE

Chapter III presented the architecture of a class of abstract machines called CONCOMs. The design specifications were the following.

1) The machine must be able to direct the control of concurrent computations whose execution must synchronize at explicit points in time.
2) The structure must be sufficiently simple so that the underlying concepts of concurrent control are easily understandable.

In Chapter IV we presented a set of CONCOM primitive instructions which are sufficient for the control of concurrent computations. It is the purpose of this chapter to illustrate the applicability of the class of CONCOMs and the proposed primitives to the problem of concurrent control. Implementations of two block structured, multitasking programming languages, STAL and SIMPAL, are presented. That is, machines (CONCOMs) are specified which interpret the control structures of multitasking languages. The task control structures of STAL and SIMPAL differ significantly and these differences are reflected in their respective implementation models. The primitive task control mechanisms of Chapter IV are shown to be sufficient to implement both models. Note that the interpreter instructions which are common to single and multiple task languages are very similar to those presented in EPL (23). They are presented here only for completeness.

The first application of the abstract machine is to the interpretation of programs written in the simple tasking language STAL. Informally,

a STAL program consists of a single block containing a declaration part and a statement part. The declarations in a block may include variables having integer or logical values and declarations of procedures, labels or locks. Statements may be assignment statements, conditional statements, procedure calls, goto statements, blocks, and task control statements. The concurrent control statements of STAL are the fork, join, lock and unlock statements discussed in Chapter II. STAL does not have arrays and restricts actual parameters to identifiers. All procedure call parameters are passed by reference as in Fortran or PL/I. The principal productions of the abstract syntax of STAL are given in Table 5.1.

The specification of a CONCOM to interpret STAL programs, a machine which will be referred to as a STAL machine, assigns meaning to the programs. The interpreter instructions are described in VDL interspersed with an English description. The reader will note that a complete definition of the STAL machine is also illustrated in Appendix I, without English description, for more compact reference.

The execution of all STAL programs starts in the initial state, IS, of a CONCOM which has but one task active and only one terminal instruction vertex. The instruction selected for execution is int-prog(t),

(ST1) int-prog(t) = int-block(t),

where t is a structure satisfying the predicate "is-prog" as defined in Table 5.1. "int-prog" is a macro-instruction which causes itself to be replaced by the instruction int-block(t), since a program is simply a block. It is defined by (ST1). Thus the execution of the instruction

Table 5.1. Syntax of STAL

```
(A1)  is-program     = is-block
(A2)  is-block       = (<s-dec-pt:is-dec-pt>,<s-st-list:is-st-list>)
(A3)  is-dec-pt      = ({<id:is-attr> | is-id(id)})
(A4)  is-attr        = is-var-attr V is-proc-attr V is-lock-attr
                       V is-label-attr
(A5)  is-var-attr    = {INT,LOG}
(A6)  is-proc-attr   = (<s-parm-list:is-id-list>,<s-st:is-st>)
(A7)  is-lock-attr   = LOCK
(A8)  is-label-attr  = LABEL
(A9)  is-st          = is-lab-st V is-unlab-st
(A10) is-lab-st      = (<s-lab:is-id>,<s-st:is-unlab-st>)
(A11) is-unlab-st    = is-assign-st V is-cond-st V is-proc-call
                       V is-goto-st V is-fork-st V is-join-st
                       V is-lock-st V is-unlock-st V is-block
(A12) is-assign-st   = (<s-lpart:is-var>,<s-rpart:is-expr>)
(A13) is-expr        = is-const V is-var V is-bin
(A14) is-const       = is-log V is-integer
(A15) is-var         = is-id
(A16) is-bin         = (<s-op1:is-expr>,<s-op2:is-expr>,
                        <s-op:is-op>)
(A17) is-op          = {+,&}
(A18) is-cond-st     = (<s-expr:is-expr>,<s-then-cl:is-st>,
                        <s-else-cl:is-st>)
(A19) is-proc-call   = (<s-id:is-id>,<s-arg-list:is-id-list>)
(A20) is-goto-st     = is-id
(A21) is-fork-st     = is-id
(A22) is-join-st     = (<s-ctr:is-var>,<s-lab:is-id>)
(A23) is-lock-st     = is-id
(A24) is-unlock-st   = is-id
```

int-prog(t) always yields a new single vertex control tree containing int-block(t).

Execution of int-block(t) -- as defined in (ST2) -- involves four actions:

1) saving the task status on the pushdown component "s-d" of the executing task;
2) creation of a new four-vertex (one terminal vertex) control tree -- for the executing task -- that will update this task's environment table and the global denotation and attribute components of the state;
3) execution of the statement list of this task in its new environment;
4) exiting the block and restoring the saved environment.

The int-block(t) is a value-returning instruction which returns a null value and modifies the dump and control component of the executing task.

Before proceeding with the formal definition, some notational conveniences are described. The three components of a task are used often and their composite selectors will have the designated shorthand that follows:

"CON" will stand for the control tree of the currently executing task and its composite selector is "s-c•s-task(S)•s-cc";

"ENV" will stand for the environment table of the currently executing task and its composite selector is "s-env•s-task(S)•s-cc";

and "DUMP" will stand for the dump component of the currently executing task and its composite selector is "s-d•s-task(S)•s-cc".

(ST2) int-block(t) =

DUMP: μ_0(<s-env:ENV(S)>,<s-c:CON(S)>,<s-d:DUMP(S)>)

CON: exit;

int-st-list(s-st-list(t));

int-dec-pt(s-dec-pt(t));

update-env(s-dec-pt(t))

After the execution of an int-block(t), the control tree of the currently executing task will have the form shown in Figure 5.1. Since execution is from the terminal vertex, the environment of this task will first be updated by the variables declared in the current block. The int-dec-pt instruction of (ST7), when executed updates the denotation and attribute tables of the state. In the new environment, the statement list of the new block is then executed by the interpreter instruction int-st-list of (ST10). The exit instruction of (ST20) will cause the reinstatement of the control, environment, and dump components (of this task) which were effective at the time of block entry.

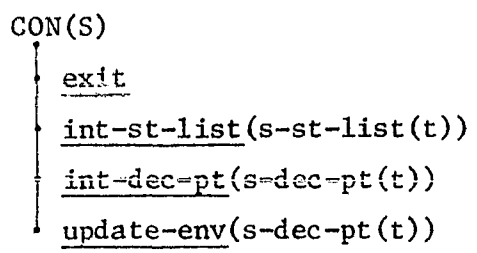

Figure 5.1. Control tree after block entry

Updating the environment by the update-env instruction,

(ST3)
```
update-env(t) =
    null;
      {update-id(id,n);
          n:un-name | id(t)≠Λ},
```

is the process of adding an entry for each variable declared in this block by the update-id instruction defined in (ST5). Execution of an update-env(t) for an argument, t, which satisfies the predicate "is-dec-pt", generates a 2m-vertex tree whose predecessor node has a null instruction. The 2m-vertices consist of m update-id(id)instructions for the m variables declared in this block and m un-name instructions (defined in (ST4)) to generate unique names for each of the declarations. The form of the control tree is given in Figure 5.2 for the declaration of the m identifiers $id_1, id_2, ---, id_m$.

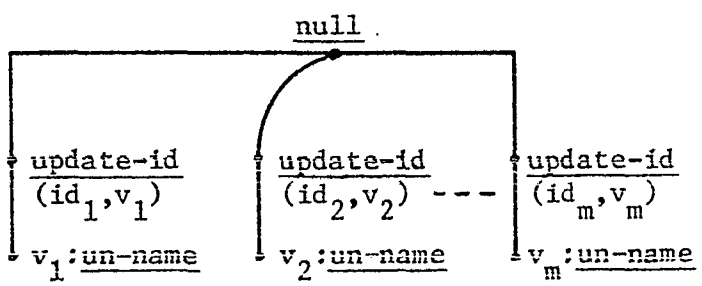

Figure 5.2. Control tree after execution of update-env(t), where the block t has m declarations

un-name,

(ST4)
$$\underline{\text{un-name}} = \quad \text{PASS: } n_{s\text{-}n(S)}, \quad \text{s-n: } s\text{-}n(S) + 1,$$

is a value-returning instruction which simply increments the unique name generator by 1 and returns the unique name whose subscript is the previous value of the component "s-d(S)" as defined in (ST4). Execution of instructions at vertices $v_1, v_2, ---, v_m$ in Figure 5.2 generate m unique names which are returned to the parameter positions $v_1, v_2, ---, v_m$ of the update-id instruction.

Execution of the instruction update-id(id,n),

(ST5)
$$\underline{\text{update-id}}(\text{id},n) = \quad \text{ENV: } \mu(\text{ENV}(S);\langle \text{id}:n\rangle),$$

simply updates the environment of the currently executing task by the selector-object pair <id:n>, where id satisfies the predicate "is-id" and n is the unique name returned by un-name. When all of the m update-id instructions have been executed, the control tree has only one terminal vertex and its instruction component is null. Execution of a null performs a no-operation function.

After deletion (execution) of update-env from the control tree, the int-dec-pt(t) instruction,

(ST6) int-dec-pt(t) =

null;

{int-dec(id,id(t),s-st-list(t))

| id(t)≠Λ

& is-id(id)},

becomes eligible for execution. Its argument, t, satisfies the predicate "is-dec-pt" and its function is to update the denotation and attribute tables, with the assumption that the environment component of the task has already been updated. Of the types of declarations considered, integer and logical do not require denotation updating; but lock, label, and procedure denotations must be initialized at block entry. All types require attribute table updating.

The int-dec-pt(t) instruction creates a control subtree with a null instruction at the root and a branch for each variable declared in this block whose vertex consists of an instruction int-dec for updating the attribute and denotation table entries as shown in (ST6). The first parameter of int-dec(id,type,st) specifies the variable name, the second parameter specifies the attributes of the declared variable, and the third specifies the statement list of the block. Note that the second argument, "id(t)", selects the attributes of declaration, which are defined by (A3) of Table 5.1.

The instruction int-dec(id,attr,st)

(ST7) int-dec(id,attr,st) =

is-var-attr(attr) → s-at: μ(s-at(S);<id•ENV(S):attr>)

```
is-proc-attr(attr) → s-at:  μ(s-at(S);<id•ENV(S):PROC>)
                     s-den: μ(s-den(S);
                              <id•ENV(S):μ0(<s-attr:attr>,
                                            <s-env:ENV(S)>)>)
is-lock-attr(attr) → s-at:  μ(s-at(S);<id•ENV(S):LOCK>)
                     s-den: μ(s-den(S);<id•ENV(S):"off">)
is-label-attr(attr) → update-lab(v,id,st);
                      v:find(id,st,1),
```

effects the updating of the attribute table for all the new unique names created for the declarations in this block. As described in (ST7) it sets denotations for lock, procedure, and label variables. But no value is set for integer and logical variables since no static initialization of variables is permitted. If the attribute parameter satisfies one of the predicates "is-proc-attr", "is-lock-attr", or "is-label-attr", then denotations which satisfy the predicates "is-proc-den", "is-lock-den", and "is-label-den", respectively, must be generated.

The environment of a procedure denotation is the current environment (after the update has taken place). Therefore, variables declared in the current block are known to the procedure. The only component of a lock denotation which is set at declaration time is its logical value. It is set to "off" since it is associated with a resource that is yet to be requested.

The label denotation has dump, environment, and statement list components which must be set at declaration time. The environment and dump components are set from the current task's environment and dump components.

But the statement list must be that list of statements starting at the specified label and all the statements following in the statement list of this block. Therefore, an instruction find(id,st,i),

(ST8)
```
find(id,st,i) =
(length(st)=i V s-lab•elem(i)•st=id)
        → PASS:  i
    T → find(id,st,i+1),
```

is executed iteratively, starting at the first statement of this block, to search for the index of the statement whose label is "id". The value returned is the index of the first such label or the index of the last statement in case the label does not exist in this block.

The instruction update-lab(n,id,st),

(ST9)
```
update-lab(n,id,st) =
s-at:  μ(s-at(S);<id•ENV(S):LABEL>)
s-den:  μ(s-den(S);μ0(<id•ENV(S):
                μ0(<s-env:ENV(S)>,<s-d:DUMP(S)>,
                   <s-st-list:
                    μ(Λ:{<elem(i):elem(j)•st
                         | i≤j≤length(st) & j≥n})>)>)),
```

sets the attribute table entry for the unique name associated with id, sets the environment and dump components from the currently active ones, and sets the statement list component to be the statement identified by "id" and all following statements in this block.

After the execution of all the int-dec, find, and update-lab instructions have been executed, the control tree contains an int-st-list(t)

instruction at its terminal vertex. The argument t satisfies the predicate "is-st-list"; and execution of the statements in this block in the order in which they fall in the list is specified by the definition of int-st-list,

(ST10)
```
int-st-list(t) =
is-Λ(t) → PASS:  Λ
      T → int-st-list(tail(t));
            int-st(head(t)).
```

This instruction (ST10) causes execution of the first statement of the list followed by the recursive execution of int-st-list for the remaining statements.

Enough basis has been laid to now consider the execution of instructions (specified by the programmer) in the environment created so far. The types of instructions possible are given in Table 5.1 in (All). In defining int-st,

(ST11)
```
int-st(t) =
is-assign-st(t) → int-assign-st(t)
is-cond-st(t)   → int-cond-st(t)
is-proc-call(t) & (at_t = PROC)
                → int-proc-call(t)
is-goto-st(t) & (lat_t = LABEL)
                → int-goto-st(t)
is-fork-st(t) & (lat_t = LABEL)
                → int-fork-st(t)
```

```
is-join-st(t) & (lat_t = LABEL)
   & (s-ctr(t)(ENV(S))•s-at(S) = INT)
                → int-join-st(t)
is-lock-st(t) & (at_t = LOCK)
                → int-lock-st(t)
is-unlock-st(t) & (at_t = LOCK)
                → int-unlock-st(t)
is-block(t)     → int-block(t),
```

assume the abbreviation "at_t" for the composite selector "s-id(t)(ENV(S))•s-at(S)" -- which selects the attributes of the procedure identifier and lock identifier "id" in a procedure call and lock or unlock statement t, respectively. Furthermore, assume the abbreviation "lat_t" denotes the selection of the label component of a fork, join or goto statement t which has the composite selector "s-lab(t)•ENV(S)•s-at(S)". Note that a procedure call, a lock and an unlock statement are valid only if the identifier specified as the procedure identifier and lock identifier, respectively, (selected by "at_t") have the PROC and LOCK declaration attributes. In the same vein -- error checking -- the label identifier of a fork, join and goto statement must have a LABEL attribute from its declaration or these statements are not legitimate. Verification that the counter specified in a join statement is an integer is accomplished before interpretation of the join.

The execution of an assignment statement involves the evaluation of the expression, which constitutes its right part, and assignment of its value to the denotation of the unique name associated with the left part, a variable, of the statement. Execution of a conditional statement

proceeds with evaluation of the logical expression followed by the execution of the then-clause if the value is true and execution of the else-clause if false. The procedure call is effected by saving the current environment, dump, and control of the currently executing task and installing the environment and control of the specified procedure. Execution of a transfer instruction must install the environment, dump, and control of the label specified in the statement. The denotation of the label contains this information. Execution of a block involves recursive execution of the int-block instruction of (ST2).

The execution of the multitasking control feature fork of STAL involves creating a new task -- with its own environment, dump, and control components -- and making this task ready for execution. Execution of a join instruction must delete the currently executing task from the system if it is not the last task to complete its computation in this concurrent computation specification. Deletion from the system involves destroying all aspects (components) of the task. Task synchronization is accommodated via execution of the lock instruction -- which moves the task requesting an unavailable resource to a blocked list for the associated resource use lock -- and the execution of the unlock instruction -- which controls allocation of the resource of the specified resource use lock.

Assuming the abbreviation "n_t" for the selector "s-lpart(t)•ENV(S)", which selects the denotation of the unique name of the identifier of the variable in the left part of an assignment statement, we define the

int-assign-st instruction,

(ST12)
```
int-assign-st(t) =
    is-var-attr(n_t(s-at(S))
        → assign(n_t,v);
            v:int-expr(s-rpart(t))
      T → error.
```

It expands into the evaluation of the right part expression by the int-expr of (ST14) followed by the assignment to n_t by assign of (ST13). Note that a valid assignment statement exists only if the attribute of n_t is integer or logical.

Execution of the assign statement,

(ST13)
```
assign(n,v) =
    s-den: μ(s-den(S);
             <n:convert (v,n(s-at(S)))>),
```
where "convert(n,v)" is a VDL primitive function which converts the value v to the representation specified in the attribute table for the unique name n,

updates the denotation of n_t by the value returned from the int-expr instruction.

Execution of int-expr(t)

(ST14)
```
int-expr(t) =
is-bin(t) → int-bin-op(s-op(t),a,b);
            a:int-expr(s-op1(t)),
            b:int-expr(s-op2(t))
```

```
is-var(t) &
     is-var-attr(n_t•s-at(S))
          → PASS:  n_t•s-at(S))
is-const(t) → PASS:  val(t)
          T → error,
```

evaluates binary operations, variables and constants whose structures are given by (A14), (A15), and (A16) of Table 5.1. The evaluation of a binary operation involves the macro expansion which evaluates both operands, which are expressions, and then the application of the operator to these values. Constant evaluation involves only returning the value of the constant; and variable evaluation involves the return of the value of the variable, in the current environment, from the denotation table. Note the "n_t" is now an abbreviation for the unique name of the variable in the current environment and is the selector "t•ENV(S)".

The application of an operator to its operands is specified by int-bin-op,

(ST15)
```
int-bin-op(op,op1,op2) =
   op = '+' → PASS:  op1 + op2
   op = '&' → PASS:  op1 & op2.
```

Interpretation of the conditional statement t is specified by int-cond-st(t),

(ST16)
```
int-cond-st(t) =
   branch(v,s-then-cl(t),s-else-cl(t));
   v:int-expr(s-expr(t)).
```

It is defined using an instruction branch(v,a,b),

(ST17) branch(v,st1,st2) =
convert(v,LOG) → int-st(st1)
T → int-st(st2)

which selects which statement, a or b, is to be executed depending upon the value v to be true or false, respectively.

Execution of a goto statement must install in the environment and dump components of the currently executing task, the environment and dump components of the block in which the label is declared. The denotation of a label (in the form given in Definition 3.4) has been set by interpreter instruction (ST9); and their assignments to the current components is necessary. Use of the abbreviations "lenv" and "ldump" for the selection of the label's environment, "s-env•lab•s-den(S)", and for the selection of the label's dump component, "s-d•lab•s-den(S)", respectively, allow for the definition of int-cond-st(t) in (ST18). Note that "lab" is an abbreviation for "s-lab(t)•ENV(S)" which selects the unique name associated with the label of the goto statement.

The specification of the control component for execution at the specified label is accomplished by placing an int-st-list(t) on the current control tree, where t is the list of statements to be executed in the block in which the label is defined. This list of statements is specified in the "s-st-list" component of the label's denotation. After execution of the block in which the label is declared, restoration of the control, environment, and dump will be accomplished by execution of the

exit statement defined in (ST19). Note that the control component must be of the form specified in Definition 3.9 and is constructed in such a manner in the int-goto-st,

(ST18) int-goto-st(t) =

ENV: lenv

DUMP: ldump

CON: μ_0(<s-in:exit>,
<succ(1):
μ_0(<s-in:int-st-list>
<s-al:μ_0(<elem(1):s-st-list•lab>)>)>).

The structure of the task n_i after execution of a goto statement is illustrated in Figure 5.3.

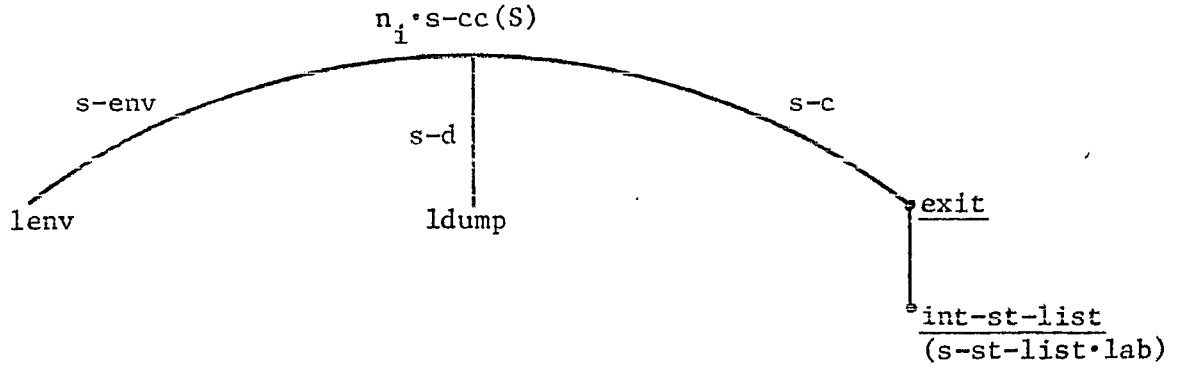

Figure 5.3. Task structure after execution of a transfer statement in task n_i

Interpretation of a procedure call involves pushing down one environment, installing a second environment, and updating the new environment with the formal/actual parameter correspondences. The parameter t of the instruction int-proc-call(t),

(ST19)
```
int-proc-call(t) =
    (length(arglist) = length(plist))
        →
    DUMP:  μ0(<s-env:ENV(S)>,
               <s-c:CON(S)>,
               <s-d:DUMP(S)>)
    ENV:   μ(env;
             {<elem(i)(plist)
               :elem(i)(arglist)(ENV(S))>
              | 1≤i≤ length(plist)})
    CON:   exit;
           int-st(st)
      T → error,
```

satisfies the predicate "is-proc-call" and has the syntax of (A17) of Table 5.1. The procedure denotation selected by "s-id", as defined in Definition 3.4, is quite complex; and introduction of the following abbreviations simplifies the definition of int-proc-call.

1) Let "n = s-id(t)•ENV(S)" denote the unique name associated with the procedure identifier of the calling procedure.
2) Let "den = n•s-den(S)" denote the selection of the procedure denotation that has the syntactic form of Definition 3.4.
3) Let "plist = s-parm-list(t)•s-attr(den)" select the list of formal parameters of the procedure denotation.

4) Let "env = s-env(den)" select the environment of the procedure denotation.

5) Let "arglist = s-arg-list(t)" select the actual parameter list of the procedure call statement.

6) Let "st = s-st•s-attr(den)" select the statement of the procedure denotation.

Execution of the int-proc-call proceeds only if the length of the actual and formal parameter lists are equal. Then the current task's components are placed in the dump; the environment of the procedure statement is installed; it is updated by the actual parameters; the control component is set to interpret the procedure statement "st"; and return from the procedure is accomplished by placing the exit statement on the control tree. Note that the updating of the environment by the actual parameters is accomplished by passing the unique name n of the actual parameter. In VDL this involves selection of the selector-object pair <id:n>, where "id" is the identifier of the actual parameter from the current environment "ENV(S)" and attaching it to the new environment "env". The control tree of the currently executing task n_i after a procedure call is illustrated in Figure 5.4.

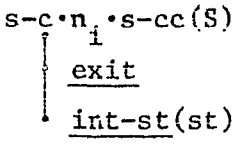

Figure 5.4. Control tree after the execution of a procedure call in task n_i

Both block entry and procedure call control structures -- as defined in (ST2) and (ST19), respectively -- save the control, environment, and dump components of the current task. The exit instruction,

(ST20)
```
exit =
    ENV:  s-env(DUMP(S))
    CON:  s-c(DUMP(S))
   DUMP:  s-d(DUMP(S)),
```

restores these components. Each component is restored from the current dump component.

In the following discussion of STAL instructions to initiate, terminate and synchronize computations, the STAL program P,

```
 1.  P:  begin  label  w1,w2,x;
 2.              integer t; lock w;
 3.          t:= 3;
 4.          fork w1;
 5.          fork w2;
 6.             st1;
 7.             lock w;
 8.             PC;
 9.             unlock w;
10.             st2;
11.         join t,x;
12.     w1:    st3;
13.            lock w;
14.            PC;
15.            unlock w;
16.            st4;
17.         join t,x;
```

```
18.      w2:    st5;
19.             join t,x;
20.      x:  REST;
21.      end,
```

will be used to illustrate such actions. In program P "sti" stands for STAL statement i and "PC" stands for program code consisting of STAL statements. The program specifies 3 concurrent computations followed by 1 computation indicated by "REST". Access to the program code PC is mutually exclusive.

In Chapter II we said that execution of the fork & statement involves the creation of a new task to be assigned to the computation specified at the label "&". And that execution of the current task is to continue at the following statement. Therefore, the expansion of int-fork-st,

(ST21)
```
int-fork-st(t) =
   init-task(v,μ0(<s-in:exit>,
                  <succ(i):μ0(<s-in:int-st-list>,
                              <s-al:st>)>),
             lenv,ldump);
      v:un-name,
```

into the task initiation primitive init-task of Definition 4.1 and the name function un-name effects the creation of a task with a unique name. Following execution of un-name and init-task, both the new task and current task are available for execution; and one of the terminal vertices of the current task's control tree specifies execution of the next statement. An illustration of the concurrent computations component of the

state is given in Figures 5.5 and 5.6 before and after initiation of a task, respectively. In Figure 5.5 interpretation of int-st(fork w1) at line 4 of program P has already been executed.

The environment and dump components of the new task are the environment and dump components of the denotation of the specified label "ℓ". The abbreviations "lenv" and "ldump" are used - as in (ST18) - to select the respective components which are passed as arguments "arg3" and "arg4" to init-task (arg1,arg2,arg3,arg4). "arg1" is the unique task name returned from un-name. "arg2" has the structure of a control tree of Definition 3.9 as illustrated in Figure 5.6. Again the abbreviation "st" is used as selector of the statement component of the label's denotation.

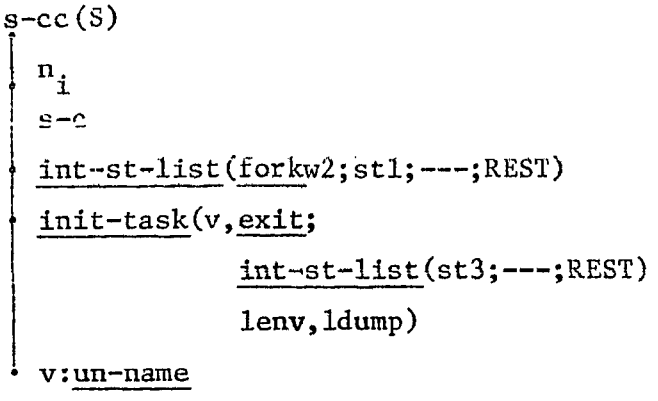

Figure 5.5. Control tree of task n_i before task initiation

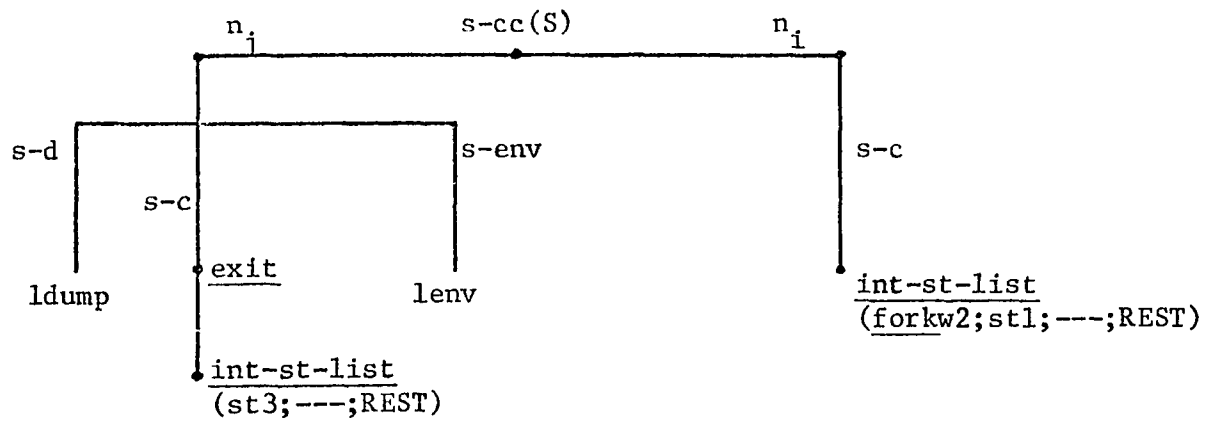

Figure 5.6. Concurrent computations component of state S after initiation of task n_j in task n_i

Execution of the int-join-st(t),

(ST22)
```
int-join-st(t) =
    int-goto-st(s-lab(t));
        terminate(s-ctr(t)•ENV(S))
```

first of all decrements the counter which indicates the number of presently active tasks; and if the counter is nonzero, the current task is terminated. The argument t satisfies the predicate "is-join-st" of (A22) in Table 5.1. The denotation of the associated counter is selected by "s-ctr(t)•ENV(S)". The execution of the task termination primitive terminate(ctr), where "ctr" is the counter's unique name, now performs the decision-making function. If the counter is zero, then the

int-goto-st(id) statement of (ST18) continues the execution of the current task at the label indicated by "id". An illustration of task termination is presented in Figure 5.7. Figure 5.7a displays the structure of the concurrent computations branch of the state of the STAL machine with the following assumptions.

1) Program P on page 65 is being executed.
2) The "join t,x" statement at lines 11, 17, and 19 has been interpreted by the int-join-st of (ST22) in the respective tasks n_i, n_j and n_ℓ.
3) The unique name assigned to the variable "t" is "n_k".

Figure 5.7b gives a snapshot of the machine state after execution of terminate (n_k) in task n_i.

Execution of the int-lock-st(t),

(ST23)
int-lock-st(t) =
request (t·ENV(S)),

assures mutually exclusive access to the program code PC enclosed by the lock w and unlock w instructions discussed in Chapter II. Since exclusive access to PC by several tasks must be granted by a scheduler, the code PC is regarded as a resource; and the associated lock, whose identifier is specified as the argument t of the int-lock-st(t) in (ST23), is regarded as the associated resource use lock. The interpretation of a lock statement is then a request to use the resource PC, and its associated lock must indicate its status. The request primitive of Definition 4.3 performs this function when the unique name of the lock in statement t is

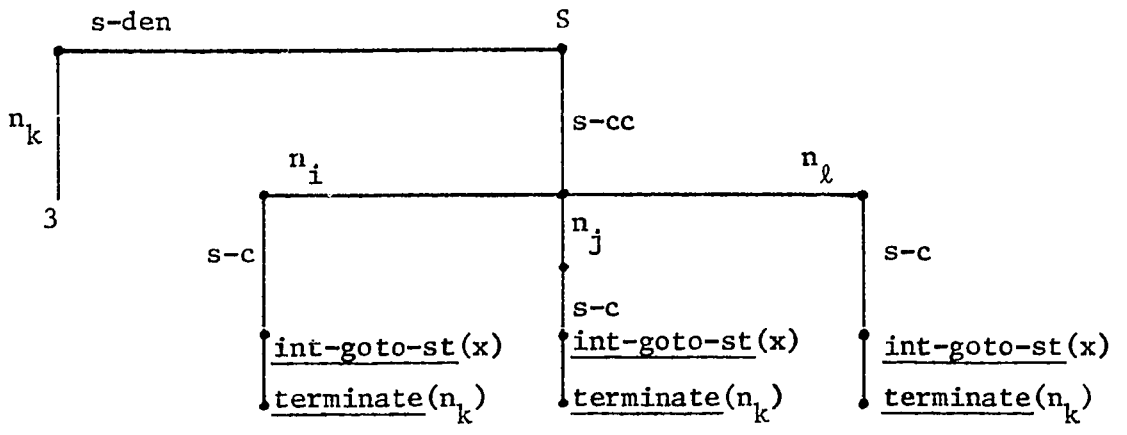

Figure 5.7a. State of the STAL machine after execution of the int-join-st in tasks n_i, n_j, and n_ℓ

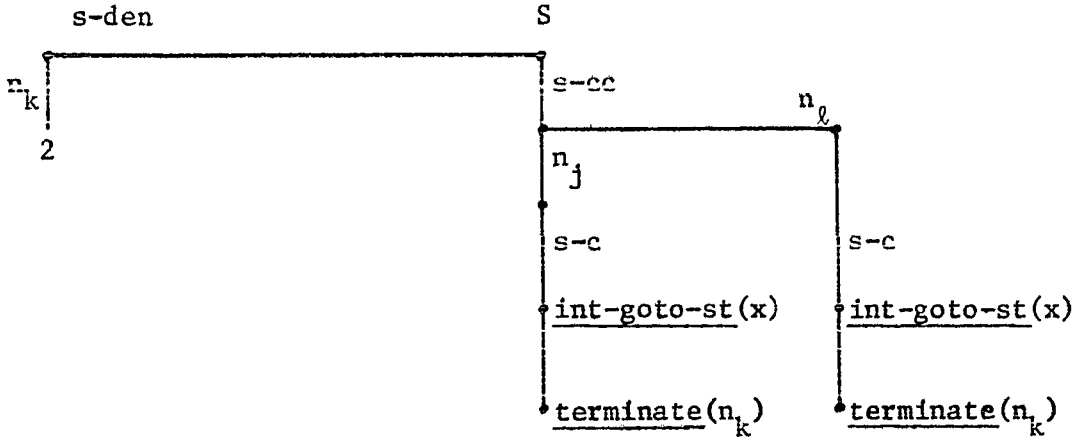

Figure 5.7b. State of STAL machine after execution of the terminate(n_k) instruction in task n_i

passed as an argument. Since the abstract syntax of a lock statement consists of only the lock identifier, "t•ENV(S)" selects its unique name.

Execution of the int-unlock-st(t) instruction,

(ST24) int-unlock-st(t) =
release(t•ENV(S))

is interpreted as the release of the resource PC. Of course, during the time that the currently active task had possession of PC, many tasks may have requested it. Therefore since the release of PC must activate only one of the tasks awaiting use of PC, a scheduler must be invoked to allocate PC. The task synchronization primitive release of Definition 4.4 performs just this function. The unique name of the lock is selected by "t•ENV(S)", as in (ST23).

Assuming the assignment of the code PC (at line 8 of program P) to a task n_i, the request for PC (at line 14 of program P) by task n_j effects the placement of n_j on the list of tasks waiting for PC. The sequence of Figures 5.8, 5.9, and 5.10 illustrate the request by n_j for PC, its waiting for access, and the release of PC by n_i, respectively. Assume the unique name of the associated resource use lock w in n_m.

The 24 interpreter instructions (ST1) through (ST24) of Appendix I and the 4 task primitive instructions of Chapter IV constitute a complete definition of the multitasking language STAL using the Vienna method. This definition assumes that programs are represented by an abstract syntax in an "intermediate language" that is independent of a specific linear representation but exhibits the operator operand structure of expressions.

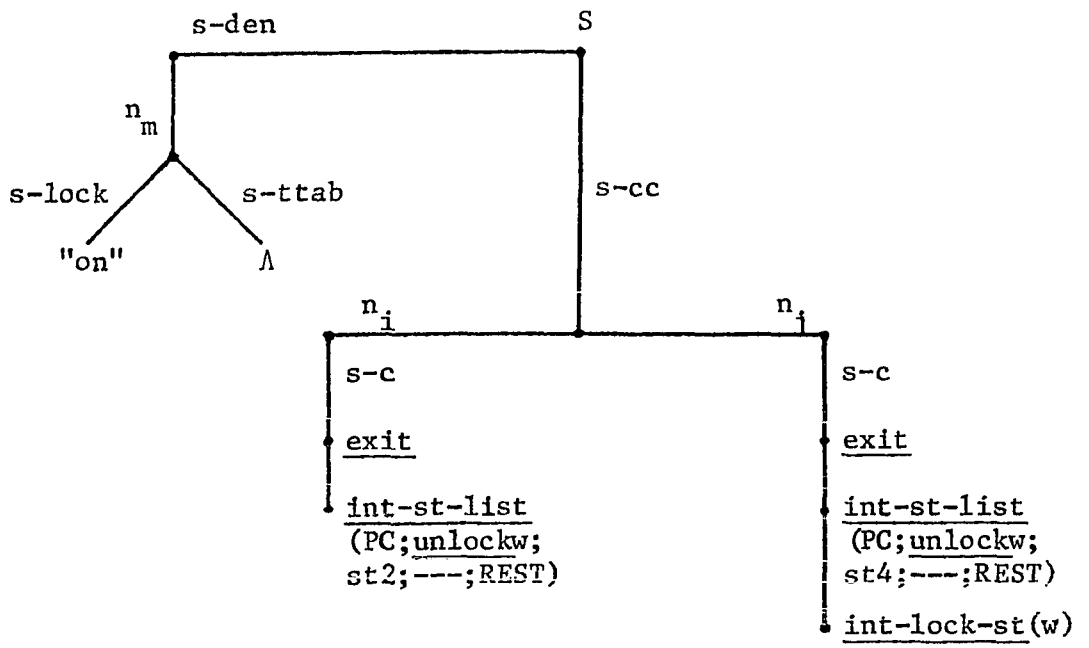

Figure 5.8. State of the STAL machine just previous to execution of program code PC in task n_i

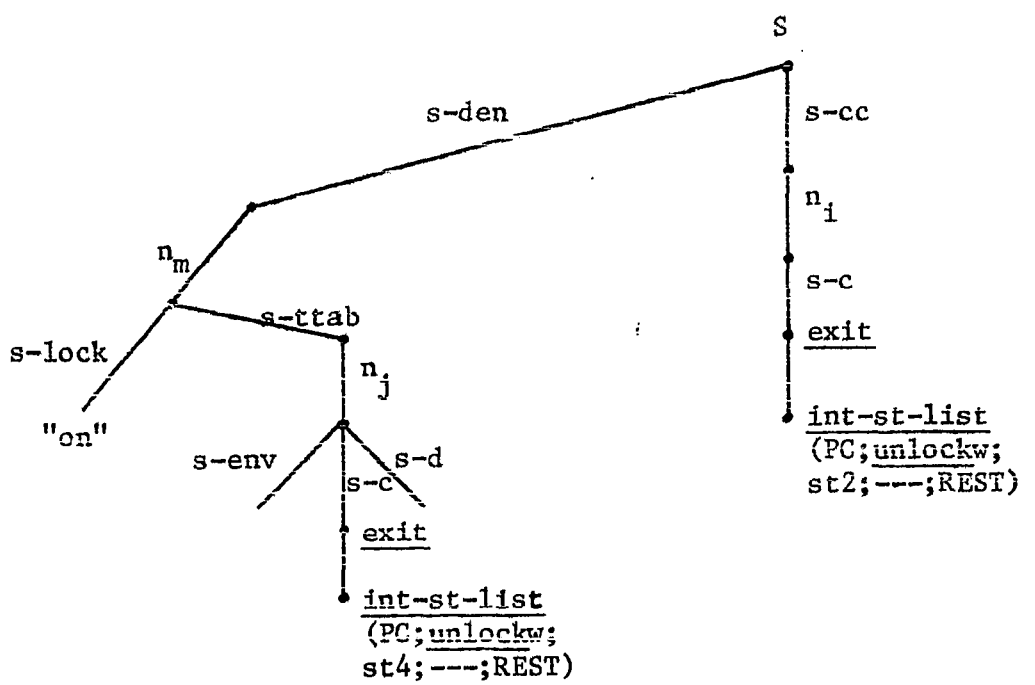

Figure 5.9. State of STAL machine after request for program code PC by task n_j

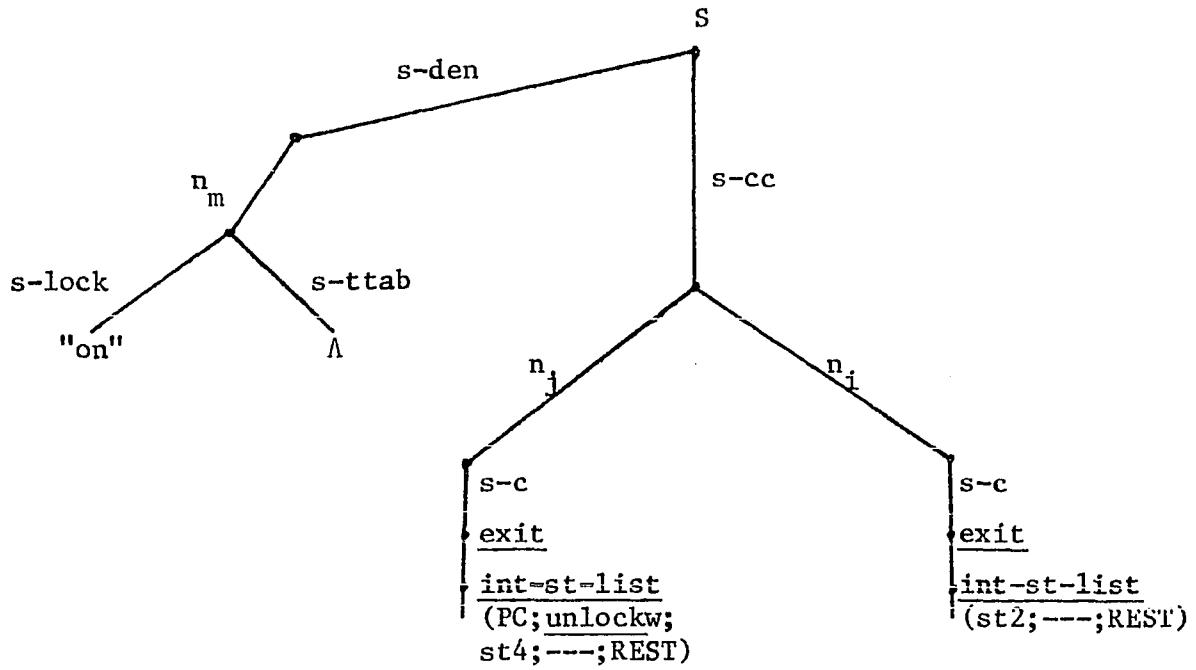

Figure 5.10. State of STAL machine after release of PC by task n_i

The semantics is defined by specifying the state transformations to which source programs give rise when they are executed in a CONCOM.

Therefore, a CONCOM has been exhibited whose structure is sufficient to control the execution of concurrent computations specified via the control structures of STAL. In order to show the generality of the CONCOM, an additional example of its application is presented. Using the same task primitive instructions, the semantics of a simple parallel processing language SIMPAL is given.

SIMPAL, like STAL, is a block structured language. The syntactic structure of the two languages differs only in the multitasking control features available to the programmer. Therefore, the state transitions in the SIMPAL machine differ from those of the STAL machine only when

specifying the interpretation of task initiation, task termination, and task synchronization. The entire syntax of SIMPAL is presented in Table 5.2 and a complete definition of the SIMPAL machine is given in Appendix II.

The initiation of m tasks in SIMPAL is specified by the special bracket pair parbegin and parend surrounding the m concurrent computations which are statements in SIMPAL. This construct has been illustrated in Chapter II. The implementation of SIMPAL will accommodate the initiation of n parallel activities during interpretation of parbegin, whereas the STAL machine initiated only 1 additional task upon interpretation of a fork.

Only 1 task is to be assigned the computation specified by the program component following a parend. As in STAL any task (the last one to complete its computation) is a candidate to continue execution; but unlike STAL, no explicit join statement is specified for each computation. The burden of termination is placed, therefore, on the implementation.

The synchronization of tasks in SIMPAL is more implicit (natural) than STAL. The critical region structures discussed in Chapter II are implemented. The statement

region v do S

implies the lock v and the resource, statement S. The statement

region v when B do S

implies conditional execution of S. This implementation will accommodate

Table 5.2. Syntax of SIMPAL

(A1')	is-program	= is-block
(A2')	is-block	= (<s-dec-pt:is-dec-pt>,<s-st-list:is-st-list>)
(A3')	is-dec-pt	= ({<id:is-attr> | is-id(id)})
(A4')	is-attr	= is-var-attr V is-proc-attr
		V is-lock-attr V is-label-attr
(A5')	is-var-attr	= {INT,LOG}
(A6')	is-proc-attr	= (<s-parm-list:is-id-list>,<s-st:is-st>)
(A7')	is-lock-attr	= LOCK
(A8')	is-label-attr	= LABEL
(A9')	is-st	= is-lab-st V is-unlab-st
(A10')	is-lab-st	= (<s-lab:is-id>,<s-st:is-unlab-st>)
(A11')	is-unlab-st	= is-assign-st V is-cond-st
		V is-proc-call V is-goto-st
		V is-par-st V is-crit-reg
		V is-cond-crit-reg V is-block
(A12')	is-assign-st	= (<s-lpart:is-var>,<s-rpart:is-expr>)
(A13')	is-expr	= is-const V is-var V is-bin
(A14')	is-const	= is-log V is-integer
(A15')	is-var	= is-id
(A16')	is-bin	= (<s-op1:is-expr>,<s-op2:is-expr>,<s-op:is-op>)
(A17')	is-op	= {+,&}
(A18')	is-cond-st	= (<s-expr:is-expr>,<s-then-cl:is-st>,
		<s-else-cl:is-st>)
(A19')	is-proc-call	= (<s-id:is-id>,<s-arg-list:is-id-list>)
(A20')	is-goto-st	= is-id
(A21')	is-par-st	= is-st-list
(A22')	is-crit-reg	= (<s-lock:is-id>,<s-st:is-st>)
(A23')	is-cond-crit-reg	= (<s-lock:is-id>,<s-expr:is-expr>,<s-st:is-st>)

the evaluation of B and execution of S as a resource protected by lock "v". Repetitive protected testing of the value of B is specified since another computation may manipulate a value in the expression.

Due to the similarity of the machines, interpreter instructions (ST1) - (ST10) and (ST12) - (ST20) of the STAL machine specify the same state transitions as instructions (SI1) - (SI10) and (SI12) - (SI20) of the SIMPAL machine, respectively. Therefore, specification of the SIMPAL machine begins with (SI11), the interpretation of a statement by int-st,

```
(SI11)   int-st(t) =
         is-assign-st(t) → int-assign-st(t)
         is-cond-st(t)   → int-cond-st(t)
         is-proc-call(t) & (at_t = PROC)
                         → int-proc-call(t)
         is-goto-st(t) & (lat_t = LABEL)
                         → int-goto-st(t)
         is-par-st(t) & is-st-list(s-st-list(t))
                         → int-par-st(t)
         is-crit-reg(t) & is-st(s-st(t))
                        & (s-lock(t)(ENV(S))•s-at(S) = LOCK)
                         → int-crit-reg(t)
         is-cond-crit-reg(t) & (s-lock(t)(ENV(S))•s-at(S) = LOCK)
                             & is-st(s-st(t)) & is-expr(s-expr(t))
                         → int-cond-crit-reg(t)
         is-block(t)     → int-block(t)
                       T → error.
```

It is important to note that the abstract syntax representation of the construction

```
parbegin
  st1;
  st2;
   .
   .
   .
  stn;
parbegin
```

specifies only a statement list. And the abstract syntax of a critical region statement specifies two components, the lock and the statement; while the conditional critical region statement specifies an expression as a third component. The interpretation of each statement type is initiated only after each component of the statement has been validated according to the syntax in Table 5.2.

The concurrent control statement in SIMPAL which creates new tasks is a sequence of statements bracketed by parbegin and parend. The compound statement is regarded as a specification of parallel activities and its structure satisfies the predicate "is-par-st". The assignment of a unique task to execute each of the parallel statements is accomplished by the int-par-st(t) instruction

(SI21)
```
int-par-st(t) =
   terminate(k);
      create-tasks(t);
         initialize(k,length(t));
            k:un-name.
```

In the following discussion the execution of the SIMPAL program Q,

```
 1.  Q:  begin  var w: shared lock;
 2.        parbegin
 3.          begin
 4.            st3;
 5.            region w do PC;
 6.            st4;
 7.          end;
 8.          st5;
 9.          begin
10.            st1;
11.            region w do PC;
12.            st2;
13.          end;
14.       parend;
15.       REST;
16.     end,
```

is illustrated in Figures 5.11, 5.12, and 5.13. It is intended to specify the same concurrent computations as program P on page 65.

The environment and dump components of each task are taken as the environment and dump components of the initiating task and selected respectively by "ENV(S)" and "DUMP(S)". The m parallel activities require the creation and assignment of m-1 tasks to m-1 parallel activities and the assignment of the current task to the last activity. The unique names for the created tasks are generated by the un-name instruction.

Every task is a candidate to continue execution of the program after completion of all the m concurrent computations. Therefore, every control tree must contain the ensuing instructions. But the placement of the terminate(k) task primitive instruction (of Definition 4.2) on the

successor node of such instructions precludes all tasks executing the ensuing statements. With the value of the unique name k set to the number of parallel activities, terminate(k) assures that only the last task to complete its computation will execute the ensuing statements. Figure 5.11 displays the state of the SIMPAL machine after assignment of the 3 tasks when initiated from task n_1. (Note that the assignment of task names to tasks is arbitrary. This particular implementation has chosen to assign the parent (initiating) task to the last computation.)

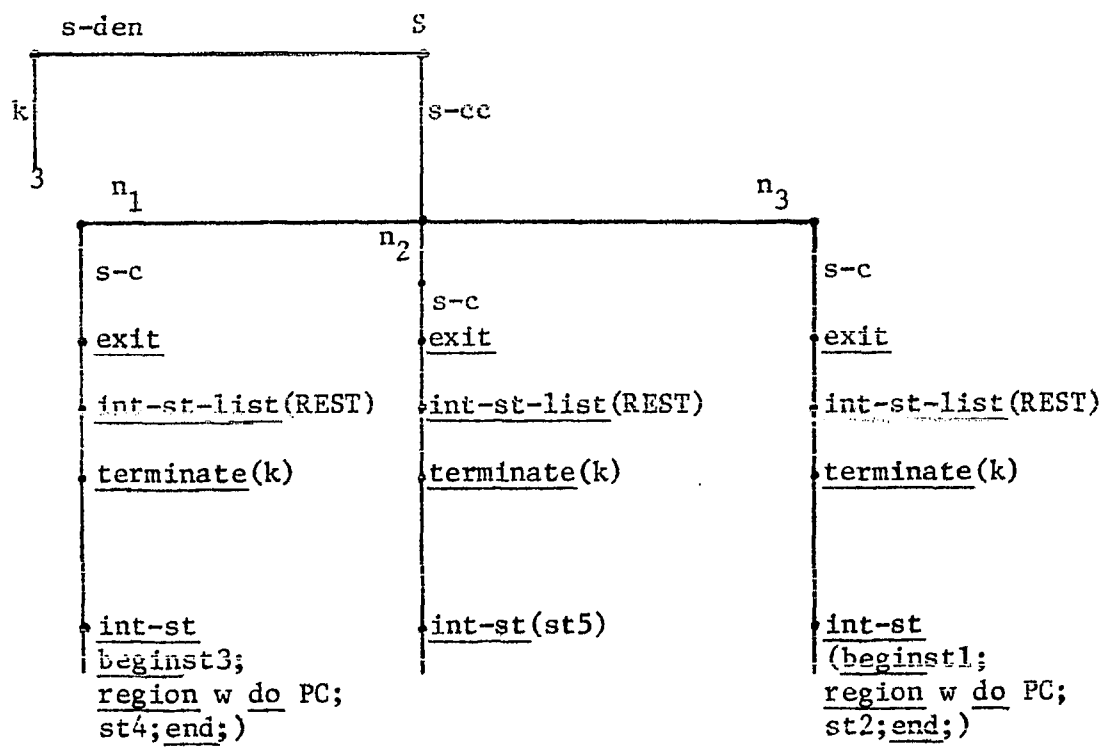

Figure 5.11. State of the SIMPAL machine after assignment of 3 tasks to the 3 concurrent computations of program Q

Creation of a denotation to contain the value of the number of parallel activities is accomplished, prior to task assignment, by invocation of un-name; and its initial value is set to the length of the parallel statement list by the instruction initialize(k,length),

(SI22)
$$\underline{\text{initialize}}(k,\ell) = \text{s-den: } \mu(\text{s-den}(S);\langle k:\ell\rangle).$$

The m-1 tasks are created and assigned by the m-1 init-task instructions generated by the instruction create-tasks,

(SI23)
$$\underline{\text{create-tasks}}(\text{stlist}) = \underline{\text{null}}; \{\underline{\text{init-task}}(v_i, \mu(\delta(S;\text{ISEL}\cdot\text{CON}(S)); \langle \text{ISEL}\cdot\text{CON}(S): \mu_0(\langle \text{s-in}:\underline{\text{int-st}}\rangle,\langle \text{s-al}:\text{elem}(i)\cdot\text{stlist}\rangle)\rangle), \text{ENV}(S),\text{DUMP}(S)); v_i:\underline{\text{un-name}} \mid 1\le i\le \text{length}(\text{stlist})-1\}, \underline{\text{int-st}}(\text{elem}(\text{length}(\text{stlist}))\cdot\text{stlist}).$$

The execution of the create-tasks instruction generates 2m-1 vertices on the control tree of the current task. m-1 of the vertices specify un-name to generate unique task names; and m-1 vertices contain init-task(arg1,arg2,arg3,arg4) instruction for assignment of new tasks. The environment and dump components, "arg3" and "arg4", respectively, are those of the current task. "arg1" is the unique name; and "arg2" has the structure of a control tree specified in Definition 3.9. It is constructed

by attaching an int-st(elem(i)(t)) instruction -- to interpret statement i of the list of parallel statements -- to the current task's control tree. Thus we include the terminate instruction and the ensuing instructions of the program on the control tree of every task. An illustration of the control tree of the initiating task n_1 is given in Figure 5.12 after execution of create-tasks at line 2 of program Q on page 78. Note the assignment of the current task n_1 to the last element of the concurrent computations by create-tasks.

Note the use of ISEL from Definition 3.11 as the control selector of the currently executing instruction. Further, note the abbreviation "CON(S) = s-c•s-task(S)•s-cc(S)" to select the current control tree. The δ-operation deletes the current instruction; and then the attachment of int-st(elem(i)•stlist) is performed. The illustration in Figure 5.11 displays the state of the SIMPAL machine after execution of the 3 init-task instructions, in Figure 5.12, in tasks n_1, n_2, and n_3.

The execution of a critical region, as specified by the instruction int-crit-reg(t) of (SI24), by a task implies exclusive access to the critical region. Therefore, the statement of the critical region is a resource protected by the lock of the critical region. Since the argument t satisfies the predicate "is-crit-region" of (A22') of Table 5.2, the statement will be selected by "s-st(t)" and the associated resource use lock will be selected by "s-lock(t)". Execution of the "int-crit-reg" instruction,

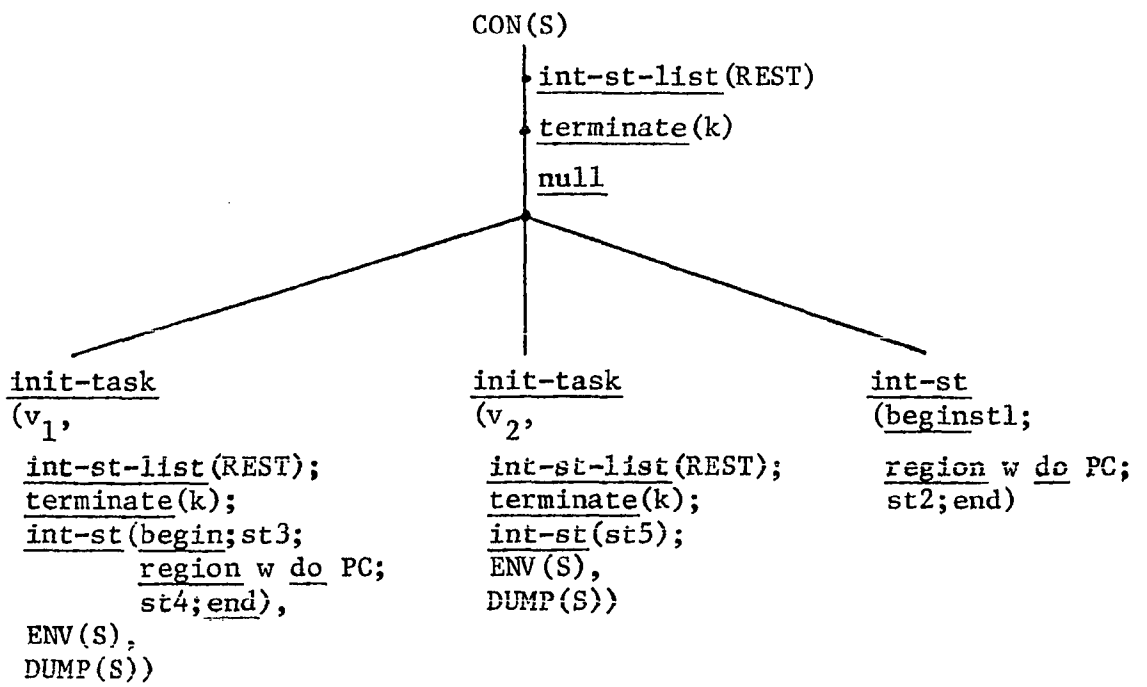

Figure 5.12. Current control tree after execution of create-tasks and 2 un-name instructions in program Q

(SI24)
```
int-crit-reg(t) =
    release(s-lock(t)•ENV(S));
        int-st(s-st(t));
            request(s-lock(t)•ENV(S)),
```

expands into the int-st(s-st(t)) instruction execution protected by the task synchronization primitives request and release to assure only one task is executing the critical statement at any one time. The execution of the control tree from its terminal vertices guarantees the request of the resource before its use.

Execution of the int-cond-crit-reg(t) instruction,

(SI25)
```
int-cond-crit-reg(t) =
   release(s-lock(t)•ENV(S));
      int-st(s-st(t));
         test(v,t);
            v:int-expr(s-expr(t));
               request(s-lock(t)•ENV(S)),
```

where t satisfies the predicate

```
is-cond-crit-reg = (<s-lock:is-id>,
                    <s-expr:is-expr>,
                    <s-st:is-st>),
```

is again interpreted as a request for resources. In this case the resources are the expression and the statement components. During evaluation of the expression and execution of the statement, the task executing the expression evaluation and the statement must have exclusive access to both the expression and the statement. But since the value of an expression cannot change unless modified by another task, whenever the value of the expression cannot be interpreted as true the resources are released and requested at a later time. The request and release tasking primitives

1) assure mutually exclusive access to the expression evaluation by the int-expr instruction,
2) the testing of the truth value of the expression by test(value) of (SI26), and
3) the execution of the statement by int-st.

The execution of test(value, conditional critical region statement),

(SI26)
```
test(value,t) =
    convert(value,LOG) → PASS:  Λ
                       T → test(v,t);
                           v:int-expr(s-expr(t));
                             request(s-lock(t)•ENV(S));
                               release(s-lock(t)•ENV(S))),
```

converts the representation of "value" to type logical. If such a representation has truth value "1", the execution of the statement component of the conditional critical region is allowed by effecting no actions in test. If it does not, then the resources have to be released and requested at a later time. "test" accomplishes this by generating a 4-vertex control subtree whose terminal vertex specifies a release instruction. The resources are later requested at the leisure of the scheduler of the associated lock due to the presence of the request primitive at the immediate predecessor node. The generation of instructions to evaluate and test the value of the expression permits repetitive checking of the expression at the convenience of the scheduler.

In order to illustrate the interpretation of critical regions, we exhibit the concurrent computations component of the SIMPAL machine in Figure 5.13 under the following assumptions.

1) Program Q on page 78 is being interpreted;
2) Interpretation has proceeded, starting from the state exhibited in Figure 5.11, with termination of task n_2 and execution of int-crit-reg in both tasks n_1 and n_3;

3) "ℓ" is the unique name associated with the lock variable "w".

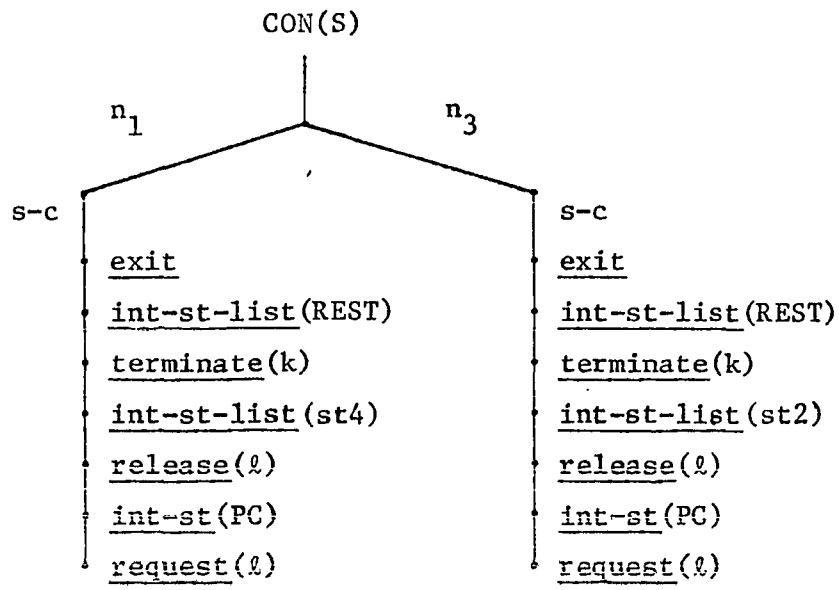

Figure 5.13. State of SIMPAL machine just prior to request for use of critical region PC

The interpretation of the critical regions now proceeds under the direction of the request and release primitives.

We have illustrated, in this chapter, two CONCOMs which define the semantics of two "simple" multitasking languages. The word "simple" is used to indicate that only those control structures of a language were implemented which seemed most pertinent in the discussion of concurrent control. Such things as input and output statements, iterative statements, and complex data types were not included as they would detract from the central theme of the dissertation. The implementation of such language features are discussed elsewhere (18) and can be implemented in a CONCOM.

The concurrent computations control structures discussed in Chapter II can all be implemented in a CONCOM, and they represent all the concurrent control structures known to the author. For example, the tasking facilities of PL/I have been implemented as a CONCOM. Its implementation defines the initiation of a procedure as a task whose

1) control component is an int-st(t), where t is the statement part of a procedure,
2) environment is that in which the procedure was declared,
3) dump component is null.

The implementation regards an event variable as a resource whose request implements a "wait" statement and whose release indicates the happening of the event.

Implementation of the counting semaphores of Dikjstra (6) require only the addition of a counter whose magnitude is the number of resources available for consumption (assignment to a task). The incrementation and decrementation of such a counter is protected from interruption by the request and release primitives.

CHAPTER VI.

DISCUSSION

The design goal of the class of abstract machines presented in this dissertation was that of conceptual clarity and not execution-time efficiency. Applicability to definition of multitasking programming languages has already been established and the Vienna definition method has been shown to be quite well suited to describe such implementations. It is therefore our purpose in this chapter to show that CONCOMs can be used as a basis for the study of computer operating systems whose function is to direct the execution of programs of many concurrent users.

As in all other areas of academic endeavor, the study of computer operating systems usually proceeds in both formal and pragmatic directions. We will discuss the validity of this model's conceptual clarity and its relationship to more efficient models in the development of both areas. CONCOMs will be applied to the five abstract areas of operating system study proposed by Denning (4).

The first area proposed is programming. This topic is concerned with language features and the operations they effect. Two implementations have been presented and it is felt that the concepts of concurrent control have been presented in a more concrete and understandable form than previous informal descriptions. CONCOMs are intended to be a framework within which to formally prove certain hypotheses about the implementation models.

The formal definition of a CONCOM in this dissertation presents a

model at a level of detail which lends itself to formal proofs of correctness of implementation, equivalence of implementations, and equivalence of interpreter models in general. We propose mathematical induction on the number of primitive task control instructions executed in a program structure as a natural method of proving correct assignment of tasks in both the STAL and SIMPAL machines. The McGowan Mapping Technique (20) seems applicable for proving equivalence of multitasking models whose concurrent control structures differ. The Twin Machine proof technique (23) certainly can be used to prove equivalence of implementation models of multitasking programming languages, such as complete to local environment, as already described by Wegner (23) for single task models. Since it is generally felt that conceptual models (like CONCOMs) of operating systems must evolve before the development of more complex and efficient models can proceed, equivalence of such models is quite important in the design of new systems.

The concepts of the four remaining abstract areas of study proposed -- storage allocation, concurrency, resource allocation, and protection -- are discussed in the framework of a CONCOM. Some needed extensions to the model are quite straightforward and are presented. The following discussion brings to the fore those concepts of current interest and the required extensions to a CONCOM to study them.

Storage allocation concerns memory management, name management, and dynamic space management. Study in this area can proceed with the addition of segment descriptor table and known segment table components to a task structure. Additional immediate state components for the active

segment table and page tables would also be necessary.

Accommodation of concurrency of tasks has been illustrated and its application to operating systems description seems quite appropos in the light of the task independence achieved. Only one problem still exists. The present model is sequential in that only one processor is allowed. But true multiprocessing can be achieved by an additional immediate component for each processor. The specification of task selection must be from the list of tasks on the concurrent computations branch, less the tasks being executed by other processors.

Resource allocation has been discussed. The current model of a CONCOM allows sharing of information among tasks through common environments. It also provides allocation of mutually exclusive resources (multiplexing). The state of deadlock exists in a CONCOM when no tasks are on the concurrent computations component and at least one task is in the task table of a resource use lock, awaiting its allocation of the resource. The set of deadlocked tasks is the union of the sets of tasks in all resource use lock task tables in the system.

Prevention of the deadlock state can be accommodated with the addition of immediate state components for the maximum claim matrix, the allocation matrix, and the available resources vector. Request and release primitives, with an additional argument containing number of resources requested and released, can implement the selection of a safe sequence (Habermann (8)) and hence allocate a new safe state.

Protection can be accommodated with the addition of an access matrix component. Additional primitive instructions must be specified to alter

such a matrix.

It is felt that this dissertation contributes to the state of the art of computing in the following manner.

1) The formalism presents the concepts of concurrent control precisely and in an understandable manner.
2) It defines the relationships of the five areas above precisely.
3) Its conceptual simplicity permits extension of its structure.
4) The existence of a formalism invites attempts to find more systematic approaches to implementation.
5) The formal description encourages formal proofs of correctness and equivalence of concurrent computation models.

BIBLIOGRAPHY

1. Anderson, J. P. Program structures for parallel processing. Communications of the Association for Computing Machinery 8, No. 12: 786-788. 1965.

2. Berry, D. M. Oregano. SIGPLAN Notices 6, No. 2: 171-190. New York, New York, Association for Computing Machinery. 1971.

3. Conway, M. E. A multiprocessor system design. American Federation of Information Processing Societies Fall Joint Computer Conference Proceedings 24: 139-146. Baltimore, Maryland, Spartan Books. 1963.

4. Denning, P. J. Third generation computer systems. Computing Surveys 3, No. 4: 175-216. 1971.

5. Dennis, J. B. and Van Horn, E. C. Programming semantics for multiprogrammed computations. Communications of the Association for Computing Machinery 9, No. 3: 143-154. 1966.

6. Dijkstra, E. W. Cooperating sequential processes. In Genuys, F., ed. Programming Languages. Pp. 43-112. New York, New York, Academic Press. 1968.

7. Fitzwater, D. R. and Schweppe, E. J. Consequent procedures in conventional computers. American Federation of Information Processing Societies Fall Joint Computer Conference Proceedings 2[illegible]. 139-146. Baltimore, Maryland, Spartan Books. 1964.

8. Habermann, A. N. Prevention of system deadlock. Communications of the Association for Computing Machinery 12, No. 7: 373-377. 1969.

9. Hansen, P. B. A comparison of two synchronizing concepts. Unpublished paper. Pittsburgh, Penn., Carnegie-Mellon University. 1971.

10. Hebalkar, P. G. Coordinated sharing of resources in asynchronous systems. Record of the Project MAC Conference on Concurrent Systems and Parallel Computations. Pp. 151-168. New York, New York, Association for Computing Machinery. 1970.

11. Hopcroft, J. E. and Ullman, J. D. Formal languages and their relation to automata. Reading, Massachusetts, Addison-Wesley Publishing Company. 1969.

12. International Business Machines Corporation. IBM System/360 Operating System PL/I (F) Programmer's guide. IBM System's Reference Library, Program Number 360S-NL-511, File No. S360-29. White Plains, New York, IBM Corporation. 1971.

13. Knuth, D. E. and McNeley, J. L. SOL-A symbolic language for general purpose systems simulation. Institute of Electrical and Electronics Engineers Transactions on Electronic Computers EC-13, No. 4: 401-408. 1964.

14. Knuth, D. E. and McNeley, J. L. A formal definition of SOL. Institute of Electrical and Electronics Engineers Transactions on Electronic Computers, EC-13, No. 4: 409-414. 1964.

15. Landin, P. J. The mechanical evaluation of expressions. Computer Journal 6, No. 4: 308-320. 1964.

16. Lee, A. N. Computer semantics -- studies of algorithms, processors, and languages. New York, New York, Van Nostrand Reinhold Company. 1972.

17. Lindsay, C. H. and Van der Meulen, S. G. Informal introduction to Algol 68. Amsterdam, Netherlands, North-Holland Publishing Company. 1971.

18. Lucas, P., Lauer, P., and Stigleitner, H. Method and notation for the formal definition of programming languages. Technical report TR 25.087. Vienna, Austria, International Business Machines Corporation. 1968.

19. McCarthy, J. M. A formal description of a subset of Algol. In Steele, T. B., ed. Formal language description languages for computer programming. Pp. 1-12. Amsterdam, Netherlands, North-Holland Publishing Company. 1966.

20. McGowan, C. L. An inductive proof technique for interpreter correctness. In Rustin, R., ed. Courant Symposium on Formal Semantics of Programming Languages. Pp. 139-147. New York, New York, Courant Institute of Mathematical Sciences. 1971.

21. Petri, C. A. Communication with automata. Supplement 1 to Technical Report RADC-TR-65-377, Vol. 1. Rome, New York, Griffiss Air Force Base. 1966.

22. Slutz, D. R. Flow graph schemata. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation. Pp. 129-141. New York, New York, Association for Computing Machinery. 1970.

23. Wegner, P. The Vienna Definition Language. Computing Surveys 1, No. 2: 5-63. 1972.

ACKNOWLEDGMENTS

I hereby acknowledge my wife, Dee, as chief "sacrificer, dishwasher, typist, editor, and sympathizer". She understands my "bugidios"; and her indignant "no's" have always meant "I understand". To my friend, colleague, kibitzer, and thesis director, Professor Charles Wright, I express my sincere appreciation for his many helpful suggestions and detailed guidance of this work. I also wish to acknowledge Professor Roy Keller, thesis director, for sharing with me his knowledge of research and for his supervision in the development of this thesis. To Professors Arthur Pohm, Robert Lambert, and Harrington Brearley, I wish to express my appreciation for the courses they taught and the friendship they extended throughout my graduate studies. Special thanks are due to Professor Robert M. Stewart who has fulfilled the role of a mentor in both my graduate studies and professional development. I also wish to thank Professor Clair Maple for his confidence throughout my graduate studies.

I wish to thank the many friends and relatives who have provided encouragement. Special thanks are due to my friends Dr. Fred Zamecnik and Mr. Robert Sharpe for the many discussions we have had.

APPENDIX I

The STAL interpreter

(ST1) int-prog(t) = int-block(t)

(ST2) int-block(t) =
DUMP: μ_0(<s-env:ENV(S)>,<s-c:CON(S)>,<s-d:DUMP(S)>)
CON: exit;
int-st-list(s-st-list(t));
int-dec-pt(s-dec-pt(t));
update-env(s-dec-pt(t))

(ST3) update-env(t) =
null;
{update-id(id,n);
n:un-name | id(t)≠Λ}

(ST4) un-name =
PASS: $n_{s\text{-}n(S)}$
s-n: s-n(S) + 1

(ST5) update-id(id,n) =
ENV: μ(ENV(S);<id:n>)

(ST6) int-dec-pt(t) =
null;
{int-dec(id,id(t),s-st-list(t))
| id(t)≠Λ
& is-id(id)}

(ST7) int-dec(id,attr,st) =
is-var-attr(attr) → s-at: μ(s-at(S);<id•ENV(S):attr>)

(ST8) find(id,st,i) =
(length(st)=i V s-lab•elem(i)•st=id)
→ PASS: i
T → find(id,st,i+1)

(ST9) update-lab(n,id,st) =
s-at: μ(s-at(S);<id•ENV(S):LABEL>)
s-den: μ(s-den(S);μ_0(<id•ENV(S):
μ_0(<s-env:ENV(S)>,<s-d:DUMP(S)>,
<s-st-list:
μ(Λ:{<elem(i):elem(j)•st
| i≤j≤length(st) & j≥n})>)>))

(ST10) int-st-list(t) =
is-Λ(t) → PASS: Λ
T → int-st-list(tail(t));
int-st(head(t))

(ST11) int-st(t) =
is-assign-st(t) → int-assign-st(t)
is-cond-st(t) → int-cond-st(t)
is-proc-call(t) & (at_t = PROC)
→ int-proc-call(t)
is-goto-st(t) & (lat_t = LABEL)
→ int-goto-st(t)
is-fork-st(t) & (lat_t = LABEL)
→ int-fork-st(t)

```
is-join-st(t) & (lat_t = LABEL)
   & (s-ctr(t)(ENV(S))•s-at(S) = INT)
                → int-join-st(t)
is-lock-st(t) & (at_t = LOCK)
                → int-lock-st(t)
is-unlock-st(t) & (at_t = LOCK)
                → int-unlock-st(t)
is-block(t)     → int-block(t)
```

(ST12)

```
int-assign-st(t) =
   is-var-attr(n_t(s-at(S))
        → assign(n_t,v);
           v:int-expr(s-rpart(t))
      T → error
```

(ST13)

```
assign(n,v) =
   s-den → μ(s-den(S);
             <n:convert (v,n(s-at(S)))>),
```

where "convert(n,v)" is a VDL primitive function which converts the value v to the representation specified in the attribute table for the unique name n

(ST14)

```
int-expr(t) =
is-bin(t) → int-bin-op(s-op(t),a,b);
             a:int-expr(s-op1(t)),
             b:int-expr(s-op2(t))
is-var(t) &
     is-var-attr(n_t•s-at(S))
          → PASS:  n_t•s-at(S))
is-const(t) → PASS:  val(t)
          T → error
```

(ST15) int-bin-op(op,op1,op2) =

op = '+' → PASS: op1 + op2

op = '&' → PASS: op1 & op2

(ST16) int-cond-st(t) =

branch(v,s-then-cl(t),s-else-cl(t));

v:int-expr(s-expr(t))

(ST17) branch(v,st1,st2) =

convert(v,LOG) → int-st(st1)

T → int-st(st2)

(ST18) int-goto-st(t) =

ENV: lenv

DUMP: ldump

CON: μ_0(<s-in:exit>,

<succ(l):

μ_0(<s-in:int-st-list>

<s-al:μ_0(<elem(l):s-st-list·lab>)>)>)

(ST19) int-proc-call(t) =

(length(arglist) = length(plist))

→

DUMP: μ_0(<s-env:ENV(S)>,

<s-c:CON(S)>,

<s-d:DUMP(S)>)

ENV: μ(env;

{<elem(i)(plist)

:elem(i)(arglist)(ENV(S))>

| 1≤i≤ length(plist)})

```
        CON:  exit;
                int-st(st)
          T → error
```

(ST20) exit =

```
        ENV:  s-env(DUMP(S))
        CON:  s-c(DUMP(S))
       DUMP:  s-d(DUMP(S))
```

(ST21) int-fork-st(t) =

```
   init-task(v,μ0(<s-in:exit>,
                  <succ(1):μ0(<s-in:int-st-list>,
                              <s-al:st>)>),
             lenv,ldump);
    v:un-name
```

(ST22) int-join-st(t) =

```
   int-goto-st(s-lab(t));
      terminate(s-ctr(t)•ENV(S))
```

(ST23) int-lock-st(t) =

```
   request (t•ENV(S))
```

(ST24) int-unlock-st(t) =

```
   release(t ENV(S))
```

APPENDIX II

The SIMPAL interpreter

(SI1) - (SI10) and (SI12) - (SI20) are the same as (ST1) - (ST10) and (ST12) - (ST20) in Appendix I.

```
(SI11)   int-st(t) =
         is-assign-st(t) → int-assign-st(t)
         is-cond-st(t)   → int-cond-st(t)
         is-proc-call(t) & (at_t = PROC)
                         → int-proc-call(t)
         is-goto-st(t) & (lat_t = LABEL)
                         → int-goto-st(t)
         is-par-st(t) & is-st-list(s-st-list(t))
                         → int-par-st(t)
         is-crit-reg(t) & is-st(s-st(t))
                        & (s-lock(t)(ENV(S))•s-at(S) = LOCK)
                         → int-crit-reg(t)
         is-cond-crit-reg(t) & (s-lock(t)(ENV(S))•s-at(S) = LOCK)
                             & is-st(s-st(t)) & is-expr(s-expr(t))
                         → int-cond-crit-reg(t)
         is-block(t)     → int-block(t)
                       T → error
```

```
(SI21)   int-par-st(t) =
            terminate(k);
               create-tasks(t);
                  initialize(k,length(t));
                     k:un-name
```

```
(SI22)   initialize(k,ℓ) =
            s-den:  μ(s-den(S);<k:ℓ>)

(SI23)   create-tasks(stlist) =
            null;
              init-task(v_i,
                        μ(δ(S;ISEL•CON(S));
                          <ISEL•CON(S):
                            μ_0(<s-in:int-st>,<s-al:elem(i)•stlist>)>),
                        ENV(S),DUMP(S));
                 v_i:un-name | 1≤i≤length(stlist)-1},
                 int-st(elem(length(stlist))•stlist)

(SI24)   int-crit-reg(t) =
            release(s-lock(t)•ENV(S));
               int-st(s-st(t));
                  request(s-lock(t)•ENV(S))

(SI25)   int-cond-crit-reg(t) =
            release(s-lock(t)•ENV(S));
               int-st(s-st(t));
                  test(v,t);
                     v:int-expr(s-expr(t));
                        request(s-lock(t)•ENV(S))

(SI26)   test(value,t) =
            convert(value,LOG) → PASS:  Λ
                               T → test(v,t);
                                   v:int-expr(s-expr(t));
                                     request(s-lock(t)•ENV(S));
                                       release(s-lock(t)•ENV(S)))
```

Greenbaum, A. L., and T. F. Slater. 1957a. Studies on the particulate components of rat mammary gland. 1. A method for determining the composition of the retained fluid. Biochem. J. 66: 148-155.

Greenbaum, A. L., and T. F. Slater. 1957b. Studies on the particulate components of rat mammary gland. 2. Changes in the levels of the nucleic acids of the mammary glands of rats during pregnancy, lactation and mammary involution. Biochem. J. 66: 155-161.

Griffith, D. R., and C. W. Turner. 1957. DNA content of mammary gland during pregnancy and lactation. Soc. Exp. Biol. Med., Proc. 95: 347-348.

Griffith, David R., and Charles W. Turner. 1959a. Desoxyribonuclease (DNAase) activity of rat mammary gland during pregnancy and lactation. Soc. Exp. Biol. Med., Proc. 97: 812-814.

Griffith, David R., and C. W. Turner. 1959b. Normal growth of rat mammary glands during pregnancy and lactation. Soc. Exp. Biol. Med., Proc. 102: 619-621.

Griffith, David R., and C. W. Turner. 1961. Normal growth of rat mammary glands during pregnancy and early lactation. Soc. Exp. Biol. Med., Proc. 106: 448-450.

Griffith, David R., and C. W. Turner. 1962. Effect of estrogen and progesterone upon milk secretion in normal and ovariectomized rats and on mammary gland DNA. Soc. Exp. Biol. Med., Proc. 110: 862-863.

Grossman, William I., and Elizabeth S. Stratton. 1969. Histochemical localization of adenosine triphosphatase in mammary myoepithelial cells. Lab. Invest. 20: 584.

Grosvenor, Clark E., and Charles W. Turner. 1960. Pituitary lactogenic hormone concentration during pregnancy in the rat. Endocrinology 66: 96-99.

Grynfeltt, Jean. 1937. Etude du processus cytologique de la secretion mammaire. Archives D'Anatomie Miscroscopique 33: 177-208.

Hamberger, L., and K. Ahren. 1964. Influence of the adrenal cortex on growth processes in the rat mammary gland. J. Endocrin. 30: 171-179.

Hamolsky, Milton, and Rhoda C. Sparrow. 1945. Influence of relaxin on mammary development in sexually immature female rats. Soc. Exp. Biol. Med., Proc. 60: 8-9.

Hamosh, Margit, and O. Scow. 1970. Plasma triglyceride and mammary adipose tissue lipase in pregnant and lactating rats. Acta Biol. Iugoslav. Ser. C. Iugoslav. Physiol. Pharmacol. Acta 6: 169-173. Original not available; abstracted in Biological Abstracts 51: 118426. 1970.

Harper, John T., Holde Puchtler, Susan N. Meloan, and Mary S. Terry. 1969. Histochemical demonstration of myoepithelial filaments (myofibrils) in tubular glomerular epithelium of human kidneys. Lab. Invest. 20: 585-586.

Hashimoto, I., and R. M. Melampy. 1967. Ovarian progestin secretion in various reproductive states and experimental conditions in the rat. Fed. Proc. 26: 485.

Hebb, Catherine, and J. L. Linzell. 1970. Innervation of the mammary gland: A histochemical study in the rabbit. Histochem. J. 2: 491-505.

Helminen, Heikki J., and Jan L. E. Ericsson. 1968a. Studies of mammary gland involution. I. On the ultrastructure of the lactating mammary gland. J. Ultrastruct. Res. 25: 193-213.

Helminen, Heikki J., and Jan L. E. Ericsson. 1968b. Studies on mammary gland involution. II. Ultrastructural evidence for auto- and heterophagocytosis. J. Ultrastruct. Res. 25: 214-227.

Helminen, Heikki J., and Jan L. E. Ericsson. 1968c. Studies on mammary gland involution. III. Alterations outside auto- and heterophagocytic pathways for cytoplasmic degradation. J. Ultrastruct. Res. 25: 228-239.

Helminen, Heikki J., and Jan L. E. Ericsson. 1968d. Studies on mammary gland involution. IV. Histochemical and biochemical observations on alterations in lysosomes and lysosomal enzymes. J. Ultrastruct. Res. 25: 240-252.

Herrenkohl, Lorraine Roth. 1971. Effects on lactation of progesterone injections administered during late pregnancy in the rat. Soc. Exp. Biol. Med., Proc. 138: 39-42.

Herrick, Earl H. 1928. The duration of pregnancy in guinea-pigs after removal and also after transplanation of the ovaries. Anat. Rec. 39: 193-200.

Hisaw, F. L., and M. X. Zarrow. 1948. Relaxin in the ovary of the domestic sow. Soc. Exp. Biol. Med., Proc. 69: 395-398.

Hisaw, Frederick L. 1926. Experimental relaxation of the pubic ligament of the guinea pig. Soc. Exp. Biol. Med., Proc. 23: 661-663.

Hollmann, K. H. 1959. L'Ultrastructure de la glande mammaire normale de la souris en lactation. J. Ultrastruct. Res. 2: 423-443.

Hollmann, K. H. 1966. Sur des aspects particuliers des proteines elaborees dans la glande mammaire. Etude au microscope electronique chez la lapine en lactation. Z. Zellforschung 69: 395-402.

Hollmann, K. H., and J. M. Verley. 1967. La regression de la glande mammaire a l'arret de la lactation. II. Etude au microscope electronique. Z. Zellforschung 82: 222-238.

Hubner, G., O. Kleinsasser, and H. J. Klein. 1969a. The fine structure of salivary duct carcinomas. On the role of myoepithelial cells in tumors of the salivary glands. Virchows Arch. Abt. Pathol. Anat. 346: 1-14.

Hubner, G., O. Kleinsasser, and H. J. Klein. 1969b. Fine structure and genessis of cylindroma (adenoidcystic carcinoma) of the salivary glands. Further investigations on the role of myoepithelial differentiated cells in salivary gland tumors. Virchows Arch. Abt. Pathol. Anat. 347: 296-315.

Izuo, Masaru, Takashi Okagaki, Ralph M. Richart, and Raffaele Lattes. 1971. Nuclear DNA content of acinar cells of the human breast during lactation. Amer. J. Clin. Pathol. 56: 443-447.

Jablonski, W. J. A., and Joseph T. Velardo. 1957a. Effects of relaxin on uterine weight of immature rats. Endocrinology 61: 474-475.

Jablonski, W. J. A., and Joseph T. Velardo. 1957b. Augmentation of estradiol-17-induced uterine growth by relaxin. Anat. Rec. 127: 423.

Jablonski, W. J. A., and Joseph T. Velardo. 1958. Uterine growth promoting action of relaxin. Soc. Exp. Biol. Med., Proc. 98: 36-37.

Jeffers, Katherine R. 1935a. Cytology of the mammary gland of the albino rat. I. Pregnancy, lactation and involution. Amer. J. Anat. 56: 257-274.

Jeffers, Katherine R. 1935b. Cytology of the mammary gland of the albino rat. II. Experimentally induced conditions. Amer. J. Anat. 56: 279-303.

Johke, Tetsu. 1971. Factors effecting the plasma prolactin level in the cow and goat as determined by radioimmunoassay. Endoc. Japon. 17: 393-401.

Johnson, Robert M., and Joseph Meites. 1958. Effects of cortisone acetate on milk production and mammary involution in parturient rats. Endocrinology 63: 290-294.

Kalra, P. S., L. Krulich, M. Quijada, S. P. Kalra, C. P. Fawcett, and S. M. McCann. 1970. Feedback effects of gonadal steroids on gonadotropins and prolactin in the rat. Excerpta Medica. Third International Congress on Hormonal Steroids 210: 53-54.

Kanematsu, Shigeto, and Charles H. Sawyer. 1962. Effects of intra-hypothalamic and intrahypophysial estrogen implants on pituitary prolactin and lactation in the rabbit. Endocrinology 72: 243-252. 1962.

Karnovsky, Morris J. 1965. A fromaldehyde-glutaraldehyde fixative of high quality for use in electron microscopy. J. Cell Biol. 27: 137A-138A.

Katzman, Philip A., David L. Larson, and Karl C. Podratz. 1971. Effects of estradiol on matabolism of vaginal tissue. Pages 107-147 in Kenneth W. McKerns, ed. The sex steroids. Appleton-Century-Crofts, New York.

Keenan, T. W., C. M. Huang, and D. James Moore. 1972. Membranes of mammary gland. III. Lipid composition of Golgi apparatus from rat mammary gland. J. Dairy Sci. 55: 51-57.

Kirkham, William R., and Charles W. Turner. 1953. Nucleic acids of the mammary glands of rats. Soc. Exp. Biol. Med., Proc. 83: 123-126.

Knox, Francis Stratton, III. 1966. Effect of relaxin, estradiol benzoate and progesterone on lactation in the rat. Unpublished Master's Thesis. Library, Iowa State University of Science and Technology, Ames, Iowa.

Kuhn, N. J. 1969. Progesterone withdrawl as the lactogenic trigger in the rat. J. Endocrin. 44: 39-54.

Kuhnel, Wolfgang. 1968. Vergleichende histologische, histochemische and elektronenmikroskopische Untersuchungen an Tranendrusen. Z. Zellforschung 87: 504-525.

Kurosumi, K., Y. Kobayashi, and N. Baba. 1968. The fine structure of mammary glands of lactating rats, with special reference to the apocrine secretion. Exp. Cell Res. 50: 177-192.

Lascelles, A. K., B. W. Gurner, and R. R. A. Coombs. 1969. Some properties of human colostral cells. Aust. J. Exp. Biol. Med. Sci. 47: 349-360.

Lee, C. S., G. H. McDowell, and A. K. Lascelles. 1968. The importance of macrophages in the removal of fat from the involuting mammary gland. Res. Vet. Sci. 10: 34-38.

Lee, J. C., and J. Hopper, Jr. 1965. Basic fuchsin-crystal violet; a rapid staining sequence for juxtaglomerular granular cells embedded in epoxy resin. Stain Tech. 40: 37-39.

Leeson, C. Roland. 1960. The histochemical identification of myoepithelium, with particular reference to the Harderian and exorbital lacrimal glands. Acta Anatomica 40: 87-94.

Leeson, C. Roland, and Thomas S. Leeson. 1970. Staining methods for sections of epon-embedded tissue for light microscopy. Canadian J. Zool. 48: 189-191.

Leeson, Thomas S., and C. Roland Leeson. 1971. Myoepithelial cells in the exorbital lacrimal and parotid glands of the rat in frozen-etched replicas. Amer. J. Anat. 132: 133-145.

Leonard, Samuel L. 1945. The relation of the placenta to the growth of the mammary gland of the rat during the last half of pregnancy. Anat. Rec. 91: 65-71.

Levy, Richard H. 1964. The effects of weaning and milk on mammary fatty acid synthesis. Biochim. Biophys. Acta 84: 229-238.

Linzell, J. L. 1952. The silver staining of myoepithelial cells, particularly in the mammary gland, and their relation to the ejection of milk. J. Anat. 86: 49-57.

Linzell, J. L. 1955. Some observations on the contractile tissue of the mammary glands. J. Physiol. 130: 257-267.

Liu, Teresa M. Y., and J. Wendell Davis. 1967. Industion of lactation by ovariectomy of pregnant rats. Endocrinology 80: 1043-1050.

Luciano, Lilliana. 1967. Die feinstruktur der Tranendruse der Ratte und Ihr Geschlechtsoimorphismus. Z. Zellforschung 76: 1-20.

Luckey, Thomas D., T. J. Mende, and J. Pleasants. 1954. The physical and chemical characterization of rat's milk. J. Nutrition 54: 345-359.

Luft, John H. 1961. Improvements in epoxy resin embedding methods. J. Biophys. Biochem. Cytol. 9: 409-414.

Lynn, Joe A. 1965. Rapid toluidine blue staining of epon-embedded and mounted "adjacent" sections. Amer. J. Clin. Pathol. 44: 57-58.

Lyons, W. R., C. H. Li, and R. E. Johnson. 1958. The hormonal control of mammary growth and lactation. Recent Progress in Hormone Research 14: 219-254.

Maeder, Leroy M. A. 1922. Changes in the mammary gland of the albino rat (Mus norvegicus albinus) during lactation and involution. Amer. J. Anat. 31: 1-26.

Mao, Peter, and Alfred Angrist. 1966. The fine structure of basal (Myoepithelial) cells of human prostate. Lab. Invest. 15: 1113.

Mayer, Gaston, and Marc Klein. 1961. Histology and cytology of the mammary gland. Pages 47-126 in S. K. Kon and A. T. Cowie, eds. Milk: the mammary gland and its' secretion. Vol. I. Academic Press, New York.

Mayne, R., and J. M. Barry. 1970. Biochemical changes during development of mouse mammary tissue in organ culture. J. Endocrin. 46: 61-70.

Meites, Joseph. 1957. Induction of lactation in rabbits with reserpine. Soc. Exp. Biol. Med., Proc. 96: 728-730.

Meites, Joseph. 1966. Control of mammary growth and lactation. Pages 669-707 in Luciano Martini and William F. Ganong, eds. Neuroendocrinology. Vol. 1. Academic Press, New York.

Meites, Joseph, and T. F. Hopkins. 1960. Induction of lactation and mammary gland growth by pituitary grafts in intact and hypophysectomized rats. Soc. Exp. Biol. Med., Proc. 104: 263-266.

Meites, Joseph, Raymond H. Kahn, and Charles S. Nicoll. 1961. Prolactin production by rat pituitary in vitro. Soc. Exp. Biol. Med., Proc. 108: 440-443.

Meites, Joseph, Charles S. Nicoll, and P. K. Talwalker. 1963. The central nervous system and the secretion and release of prolactin. Pages 238-277 in Andrew V. Nalbandov, ed. Advances in neuroendocrinology. University of Illinois Press, Urbana, Illinois.

Meites, Joseph, and James T. Sgouris. 1953. Can the ovarian hormones inhibit the mammary response to prolactin? Endocrinology 53: 17-23.

Meites, Joseph, P. K. Talwalker, and C. S. Nicoll. 1960. Initiation of lactation in rats with hypothalamic or cerebral tissue. Soc. Exp. Biol. Med., Proc. 103: 298-300.

Meites, Joseph, and C. W. Turner. 1942a. Studies concerning the mechanism controlling the initiation of lactation at parturition. I. Can estrogen supress the lactogenic hormone of the pituitary? Endocrinologie 30: 711-718.

Meites, Joseph, and C. W. Turner. 1942b. Studies concerning the mechanism controlling the initiation of lactation at parturition. II. Why lactation is not initiated during pregnancy. Endocrinology 30: 719-725.

Meites, Joseph, and C. W. Turner. 1942c. Studies concerning the mechanism controlling the initiation of lactation at parturition. III. Can estrogen account for the precipitous increase in the lactogen content of the pituitary following parturition? Endocrinology 30: 726-733.

Meites, Joseph, and C. W. Turner. 1942d. Effect of estrone on lactogen content in pituitary and blood of male rabbits. Soc. Exp. Biol. Med., Proc. 49: 190-193.

Meites, Joseph, and C. W. Turner. 1942e. Lactogenic content of pituitaries of pseudopregnant rabbits. Soc. Exp. Biol. Med., Proc. 49: 193-194.

Meites, Joseph, and C. W. Turner. 1948. Studies concerning the induction and maintenance of lactation. Missouri University Agricultural Experiment Station Research Bulletin 415.

Millonig, G. J. 1961. Advantages of a phosphate buffer for OsO_4 solution in fixation. Journal of Applied Physics 32: 1637.

Minaguchi, Hiroshi, James A. Clemens, and Joseph Meites. 1968. Changes in pituitary prolactin levels in rats from weaning to adulthood. Endocrinology 82: 555-558.

Moon, R. C., D. R. Griffith, and C. W. Turner. 1959. Normal and experimental growth of rat mammary gland. Soc. Exp. Biol. Med., Proc. 101: 788-790.

Moon, Richard C., and Charles W. Turner. 1960. Thyroid hormone and mammary gland growth in the rat. Soc. Exp. Biol. Med., Proc. 103: 149-151.

Murad, Tariq M. 1970. Ultrastructural study of rat mammary gland during pregnancy. Anat. Rec. 167: 17-35.

Murad, Tariq M., and Emmerich von Haam. 1967. Ultrastructural study of myoepithelial cells of the human mammary gland. Lab. Invest. 16: 667-678.

Nelson, Warren O. 1935. The effect of hypophysectomy upon mammary gland development and function in the guinea pig. Soc. Exp. Biol. Med., Proc. 33: 222-224.

Nelson, Warren O. 1937. Studies on the physiology of lactation. Amer. J. Anat. 60: 341-365.

Nicoll, Charles S., and Joseph Meites. 1962. Estrogen stimulation of prolactin production by rat adenohypophysis *in vitro*. Endocrinology 70: 272-277.

Nicoll, Charles S., P. K. Talwalker, and Joseph Meites. 1960. Initiation of lactation in rats by nonspecific stresses. Amer. J. Physiol. 198: 1103-1106.

Nishimura, Toshio, and Yukio Manabe. 1967. Oxytocin sensitivity and effects of estrogen and progesterin on metreynter induced abortions at mid-pregnancy: Preliminary report. Amer. J. Obstet. Gynecol. 98: 1087-1090.

Okada, Mitsuo. 1956a. Histology of the mammary gland. I. Cytological and cytochemical studies of colostrum bodies appearing in mammary glands of pregnant, lactating and post-weaned mice. Tohoku J. Agr. Res. 7: 35-49.

Okada, Mitsuo. 1956b. Histology of the mammary gland. II. Effects of stagnant milk on the colostrum bodies in the mammary glands of rats. Tohoku J. Agr. Res. 7: 115-129.

Okada, Mitsuo. 1957. Histology of the mammary gland. III. Wandering cells in mammary tissue at the farrowing and weaning stage and their relation to circulating blood leucocytes in mice. Tohoku J. Agr. Res. 8: 121-127.

Okada, Mitsuo. 1958a. Histology of the mammary gland. IV. Comparative morphology of the degenerative lymphoid cells in the mammary tissues, lymphoid organs and gut of mice. Tohoku J. Agr. Res. 9: 1-21.

Okada, Mitsuo. 1958b. Histology of the mammary gland. V. Effect of ACTH on the lymphoid cell counts in the mammary gland of lactating mice and rats. Tohoku J. Agr. Res. 9: 23-35.

Ota, Katuaki. 1964. Mammary involution and engorgement after arrest of suckling in lactating rats indicated by the contents of nucleic acids and milk protein of the gland. Endocrin. Japon. 11: 146-152.

Ott, Isaac, and John C. Scott. 1910. The action of infundibulin upon mammary secretion. Soc. Exp. Biol. Med., Proc. 8: 48-49.

Paape, Max J., and Claude Desjardins. 1971. Nursing duration and suckling intensity: Effects on plasma corticosterone, circulating leukocytes and mammary nucleic acids. Soc. Exp. Biol. Med., Proc. 138: 12-17.

Palade, G. E. 1952. A study of fixation for electron microscopy. J. Exp. Med. 95: 285-298.

Petersen, W. E. 1942. Effect of certain hormones and drugs on the perfused mammary gland. Soc. Exp. Biol. Med., Proc. 50: 298-300.

Pinto, Roberto Martin, Ubico Lerner, and Herminia Pontelli. 1967. The effect of progesterone on oxytocin-induced contraction of the three separate layers of human gestational myometrium in the uterine body and lower segment. Amer. J. Obstet. Gynecol. 98: 547-554.

Pool, Charlotte R. 1969. Hematoxylin-eosin staining of OsO_4-fixed epon-embedded tissue; prestaining oxidation by acidified H_2O_2. Stain Tech. 44: 75-79.

Pose, S. V., and C. Fielitz. 1961. The effects of progesterone on the response of the pregnant human uterus to oxytocin. Pages 229-239 in R. Caldeyro-Barcia and H. Heller, eds. Oxytocin. Pergamon Press, New York.

Puchtler, Holde, Faye Sweat, and John J. Sesta. 1966. Azophloxine GA, a selective stain for light and fluorescense microscopy of myoepithelial cells. Stain Tech. 41: 15-17.

Puchtler, Holde, Faye Sweat, Mary S. Terry, and H. M. Conner. 1969. Investigation of staining, polarization and fluorescense-microscopic properties of myoendothelial cells. J. Microscopy 89: 95-104.

Ramirez, V. D., and S. M. McCann. 1964. Induction of prolactin secretion by implants of estrogen into the hypothalamo-hypophysial region of female rats. Endocrinology 75: 206-214.

Ratner, Albert, P. K. Talwalker, and Joseph Meites. 1963. Effect of estrogen administration in vivo on prolactin release by rat pituitary in vitro. Soc. Exp. Biol. Med., Proc. 112: 12-15.

Reece, R. P., and J. A. Bivins. 1942. Progesterone effect on pituitary content and on mammary glands of ovariectomized rats. Soc. Exp. Biol. Med., Proc. 49: 582-584.

Richard, Ph., I. Urban, and R. Denamur. 1970. The role of the dorsal tracts of the spinal cord and of the mesencaphalic and thalamic lemniscal system in the milk-ejection reflex during milking in the ewe. J. Endocrin. 47: 45-53.

Richards, R. C., and G. K. Benson. 1971a. Ultrastructural changes accompanying involution of the mammary gland in the albino rat. J. Endocrin. 51: 127-135.

Richards, R. C., and G. K. Benson. 1971b. Structural changes associated with inhibition of involution of the mammary gland in the albino rat. J. Endocrin. 51: 137-148.

Richards, R. C., and G. K. Benson. 1971c. Involvement of the macrophage system in the involution of the mammary gland in the albino rat. J. Endocrin. 51: 149-156.

Richardson, K. C. 1949. Contractile tissues in the mammary gland, with special reference to myoepithelium in the goat. Proc. Royal Soc. London, Ser. B, 136: 30-45.

Richardson, K. C., L. Jarett, and E. H. Finke. 1960. Embedding in epoxy resins for ultrathin sectioning in electron microscopy. Stain Tech. 35: 313-323.

Roberts, John S. 1971. Progesterone-inhibition of oxytocin release during vaginal distention: Evidence for a central site of action. Endocrinology 89: 1137-1141.

Sar, M., and Joseph Meites. 1968. Effects of progesterone, testosterone, and cortisol on hypothalamic prolactin inhibitory factor and pituitary prolactin content. Soc. Exp. Biol. Med., Proc. 127: 426-429.

Sawyer, Wilbur H., and Edward H. Frieden. 1952. In vitro inhibition of spontaneous contractions of the rat uterus by relaxin-containing extracts of sow ovaries. Anat. Rec. 113: 566.

Scott, Bronnetta, and Daniel C. Pease. 1959. Electron microscopy of the salivary and lacrimal glands of the rat. Amer. J. Anat. 104: 115-161.

Selye, H. 1934. On the nervous control of lactation. Amer. J. Physiol. 107: 535-548.

Shani, J. (Mishkinsky), L. Zanbelman, K. Khazen, and F. G. Sulman. 1970. Mammotrophic and prolactin-like effects of rat and human placentae and amniotic fluid. J. Endocrin. 46: 15-20.

Shires, T. K., M. Johnson, and K. M. Richter. 1969. Hematoxylin staining of tissues embedded in epoxy resins. Stain Tech. 44: 21-25.

Silver, I. A. 1954. Myoepithelial cells in the mammary and parotid glands. J. Physil. 129: 89-99.

Slater, T. F. 1962. Studies on mammary involution. I. Chemical changes. Archives Internationales de Physiologie et de Biochimie 70: 167-178.

Spurlock, Ben O., Margaret S. Skinner, and Anthony A. Kattine. 1966. A simple rapid method for staining epoxy-embedded specimens for light microscopy with the polychromatic stain Paragon-1301. Amer. J. Clin. Pathol. 46: 252-258.

Stein, Olga, and Yechezkiel Stein. 1967. Lipid synthesis, intracellular transport, and secretion. II. Electron microscopic radioautographic study of the mouse lactating mammary gland. J. Cell Biol. 34: 251-263.

Talwalker, P. K., C. S. Nicoll, and J. Meites. 1961. Induction of mammary secretion in pregnant rats and rabbits by hydrocortisone acetate. Endocrinology 69: 802-808.

Tamarin, Arnold. 1966. Myoepithelium of the rat submaxillary gland. J. Ultrastruct. Res. 16: 320-338.

Traurig, Harold H. 1967. A radiographic study of cell proliferation in the mammary gland of the pregnant mouse. Anat. Rec. 159: 239-248.

Traurig, Harold H., and Charles F. Morgan. 1964. Autoradiographic studies of the epithelium of mammary gland as influenced by ovarian hormones. Soc. Exp. Biol. Med., Proc. 115: 1076-1088.

Travill, A. A., and M. F. Hill. 1963. Histochemical demonstration of myoepithelial cell activity. Quart. J. Exp. Physiol. 48: 423-428.

Trentin, J. J. 1951. Relaxin and mammary growth in the mouse. Soc. Exp. Biol. Med., Proc. 78: 9-11.

Trump, Benjamin F., Edward A. Smuckler, and Earl P. Benditt. 1961. A method for staining epoxy sections for light microscopy. J. Ultrastruct. Res. 5: 343-348.

Tucker, H. Allen, and Ralph P. Reece. 1962. Nucleic acid estimates of mammary tissue and nuclei. Soc. Exp. Biol. Med., Proc. 111: 639-642.

Tucker, H. Allen, and Ralph P. Reece. 1963a. Nucleic acid content of mammary glands of pregnant rats. Soc. Exp. Biol. Med., Proc. 112: 370-372.

Tucker, H. Allen, and Ralph P. Reece. 1963b. Nucleic acid content of mammary glands of lactating rats. Soc. Exp. Biol. Med., Proc. 112: 409-412.

Tucker, H. Allen, and Ralph P. Reece. 1964. Nucleic acid content of suckled and non-suckled mammary glands of lactating rats. Soc. Exp. Biol. Med., Proc. 115: 887-890.

Turkington, Roger W. 1971a. Hormonal control of lactose synthetase in developing mammary gland. Pages 49-59 in Max Hamburg and E. J. W. Barrington, eds. Hormones in development. Academic Press, New York.

Turkington, Roger W. 1971b. Hormonal regulation of mammary gland development in vitro. Pages 383-410 in Kenneth W. McKerns, ed. The sex steroids. Appleton-Century-Crofts, New York.

Turkington, Roger W., Keith Brew, Thomas C. Vanaman, and Robert L. Hill. 1968. The hormonal control of lactose synthetase in the developing mouse mammary gland. J. Biol. Chem. 243: 3382-3387.

Turkington, Roger W., and Marie Riddle. 1970. Hormone-dependent formation of polysomes in mammary cells in vitro. J. Biol. Chem. 245: 5145-5152.

Turner, C. W. 1952. The Mammary Gland. Lucas Brothers, Columbia, Missouri.

Turner, C. W., and W. D. Cooper. 1941. Assay of posterior pituitary factors which contract the lactating mammary gland. Endocrinology 29: 320-323.

Venable, John H., and Richard Coggeshall. 1965. A simplified lead citrate stain for use in electron microscopy. J. Cell Biol. 25: 407-408.

Verley, J. M., and K. H. Hollmann. 1967. La regression de la glande mammaire a l'arret de la lactation. I. Etude au microscope optique. Z. Zellforschung 82: 212-221.

Du Vigneaud, V., C. Ressler, J. M. Swan, C. W. Robers, P. G. Katsoyannis, and S. Gordon. 1953. The synthesis of an octapeptide amide with the hormonal activity of oxytocin. J. Amer. Chem. Soc. 75: 4879-4883.

Wada, Hiroshi, and Charles W. Turner. 1958. Role of relaxin in stimulating mammary gland growth in mice. Soc. Exp. Biol. Med., Proc. 99: 194-197.

Wada, Hiroshi, and Charles W. Turner. 1959a. Effect of relaxin on mammary gland growth in female mice. Soc. Exp. Biol. Med., Proc. 101: 707-709.

Wada, Hiroshi, and Charles W. Turner. 1959b. Effect of relaxin on mammary gland growth in the female rat. Soc. Exp. Biol. Med., Proc. 102: 568-570.

Wada, Hiroshi, and Charles W. Turner. 1963a. Role of relaxin in pregnancy maintenance and termination in mice. Soc. Exp. Biol. Med., Proc. 113: 631-634.

Wada, Hiroshi, and Charles W. Turner. 1963b. Influence of progesterone, estradiol benzoate, and relaxin upon placentomata formation in mice. Soc. Exp. Biol. Med., Proc. 113: 635-637.

Watson, Michael L. 1958. Staining of tissue sections for electron microscopy with heavy metals. J. Biophys. Biochem. Cytol. 4: 475-478.

Waugh, Douglas, and Ellen van der Hoeven. 1962. Fine structure of the human adult female breast. Lab. Invest. 11: 220-228.

Weatherford, Harold L. 1929. A cytological study of the mammary gland: Golgi apparatus, trophospongium and other cytoplasmic canaliculi, mitochondria. Amer. J. Anat. 44: 199-281.

Weiss, Leon. 1972. The Cells and Tissues of the Immune System. Prentice-Hall, Inc., New Jersey.

Wellings, S. R., and K. B. DeOme. 1961. Milk protein droplet formation in the Golgi apparatus of the C3H/Crgl mouse mammary epithelial cells. J. Biophys. Biochem. Cytol. 9: 479-485.

Wellings, S. R., K. B. DeOme, and D. R. Pitelka. 1960. Electron microscopy of milk secretion in the mammary gland of the C3H/Crgl mouse. I. Cytomorphology of the prelactating and lactating gland. J. U.S. Natl. Cancer Inst. 25: 393-421.

Wislocki, George B., Leon P. Weiss, Mario H. Burgos, and Richard A. Ellis. 1957. The cytology, histochemistry and electron microscopy of the granular cells of the metrial gland of the gravid rat. J. Anat. 91: 130-140.

Yamauchi, Akio, and Geoffrey Burnstock. 1967. Nerve-myoepithelium and nerve glandular epithelium contacts in the lacrimal gland of the sheep. J. Cell Biol. 34: 917-919.

Yoshinaga, K., R. A. Hawkins, and J. F. Stocker. 1969. Estrogen secretion by the rat ovary in vivo during the estrous cycle and pregnancy. Endocrinology 85: 103-112.

Zarzycki, Jan, Alina Peryt, Barbara Klubinska, Teresa Hojac, and Krystyna Zak. 1969. Histochemische Untersuchungen uber den Involutionmechanismus der Milchdruse. Histochemie 18: 314-320.

ACKNOWLEDGEMENTS

I would like to gratefully acknowledge the assistance of my major professor, Dr. David R. Griffith, for his patience, help and understanding during my tenure as his graduate student. I would also like to in general thank my graduate committee for their generous help in directing my research and for their assistance in the preparation of this manuscript. I would like to specifically acknowledge the use, by Drs. Harry T. Horner, Jr. and Charles J. Ellis, of their microscope facilities. I would also like to express my appreciation to the Department of Zoology and Entomology for the financial support given me toward my research. Finally I wish to thank my family and friends for their support and encouragement during these past years.